VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Jazyk SWIFT 3.0 a jeho možnosti

Language SWIFT 3.0 and its Properties

2017 Bc. Adam Zikmund

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Adam Zikmund**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Jazyk SWIFT 3.0 a jeho možnosti
Language SWIFT 3.0 and its Properties

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je vytvořit palikaci pro mobilní telefon iPhone v jazyce SWIFT 3.0 za účelem zhodnocení jeho možností a porovnání s jazykem Objective-C.

Práce bude obsahovat:
1. Popis jazyka SWIFT 3.0 a technologií na nihž je vystavěn.
2. Porovníní vlastností s jazykem Objective-C.
3. Implementace ukázkové aplikace v jazyce SWIFT 3.0 a Objective-C.
4. Porovnání obou řešení v oblastech běžného kódu, komponent uživatelského rozhraní, grafického rozhraní, přístup k hardwaru telefonu z hlediska:
a) rychlosti,
b) složitosti,
c) programátorské přívětivosti.

Seznam doporučené odborné literatury:

[1] Apple Inc. The Swift Programming Language (Swift 3 beta): Swift Programming Series. Apple Inc., 2015.
[2] Apple Inc. Using Swift with Cocoa and Objective-C (Swift 3 beta): Swift Programming Series. Apple Inc., 2015.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. dubna 2017 .....

Rád bych na tomto místě poděkoval panu doktoru Davidu Ježkovi, který mou diplomovou práci vedl a byl mi nápomocen při její vypracovávání.

Abstrakt

Tato práce se zaměřuje na problematiku nového programovacího jazyka Swift. Hlavním smyslem je popis vlastností a chování jazyka a také jeho srovnání se starším programovacím jazykem Objective-C. Jazyky jsou porovnány podle rychlosti, složitosti a také programátorské přívětivosti. K dosažení těchto cílů byla vytvořena mobilní aplikace, která implementuje jednotlivé funkcionality pomocí obou jazyků a následně lze díky ní porovnat jednotlivá řešení.

Klíčová slova: Swift, Objective-C, Apple

Abstract

This thesis is focused on the issue of a new programming language Swift. The main purpose of this thesis is description of the characteristics and behaviour of the language and its comparsion with older programming language Objective-C. Languages are compared in terms of speed, complexity and programmer experience. To achieve these objectives was created a mobile application which implments fuctionalities using both languages and then compares its individual solutions.

Key Words: Swift, Objective-C, Apple

Obsah

Seznam použitých zkratek a symbolů

WWDC	–	Worldwide Developers Conference
ARC	–	Automatic Reference Counting
LLVM	–	Low Level Virtual Machine
GCD	–	Grand Central Dispatch
API	–	Application Programming Interface
ARC	–	Automatic Reference Counting
GPS	–	Global Positioning System
CD	–	Compact Disc

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Společnost Apple hraje v dnešním světě moderních technologií dominantní úlohu. Firma jako taková, se snaží udávat trendy ve všech ohledech, které dnešní doba poskytuje. Ať se jedná o mobilní telefony či přenosné počítače. Tak jako tak se ostatní technologické společnosti snaží v mnoha ohledech Apple napodobit. Aby všechny tyto moderní zařazení bylo možné v dnešní době smysluplně používat, je potřeba pro ně mít vytvořeny kvalitní aplikace. Díky odkazu a základní filozofii Steva Jobse, Apple vždy kladl důraz na spojení hardwaru se softwarem a jinak tomu není ani dnes pod vedením Tima Cooka. To mohl být také jeden zdůvodů, proč tato firma představila v roce 2014 na WWDC nový programovací jazyk Swift, který slouží k programování aplikací pro všechny operační systémy Applu.

Jazyk Swift společnost Apple představila jako moderní, bezpečný, a hlavně rychlý jazyk, který měl nastolit nové trendy v oblasti vývoje mobilních a desktopových aplikací. Po třech letech od vydání první verze lze s jistotou říci, že se to firmě povedlo. Důkazem je poměrně rychlá adopce mezi vývojáři, kteří přecházejí ze staršího Objective-C na nový Swift. Kromě moderních konstrukcí, měl jazyk Swift přinést také vyšší rychlost oproti jazyku Objective-C. Podle vyjádření Applu, měl Swift v určitých situacích převyšovat v rychlosti i jazyk C++.

Všechna tato předsevzetí vedla k vytvoření diplomové práce, která si klade za cíl, všechny tyto aspekty ověřit. K ověření těchto aspektů bude vytvořená mobilní aplikace, která prakticky porovná hlavní rozdíly mezi jazyky Swift a Objective-C. Dále pak porovná rychlost, syntaktickou složitost a uživatelskou přívětivost obou jazyků. Tato porovnávání budou v aplikaci testována na základních softwarových konstrukcích, přístupech k hardwaru a také na práci s grafickým rozhraním.

Práce je logicky členěná do kapitol, které postupně představují jazyk Swift ve verzi 3.0. Přičemž první kapitola shrne představení programovacího jazyka Swift společností Apple. Dále pak podrobně popíše novinky v jednotlivých verzích jazyka Swift. Následující kapitola se následně zaobírá interoperabilitou jazyků Objective-C a Swift. V této kapitole jsou zmíněny nejzásadnější procesy, ke kterým dochází při přemosťování obou programovacích jazyků. Třetí kapitola popisuje vlastnosti a chování jazyka Swift. Tato kapitola je členěná na jednotlivé oblasti, které jsou pro popis programovacího jazyka stěžejní. Mezi tyto oblasti patří kupříkladu třídy, vlastnosti, výjimky, správa paměti a jiné. V další kapitole je pak popsaná praktická část práce, která řeší porovnávání obou jazyků. Rovněž i tato kapitola, je rozčleněna do několika sekcí, které postupně srovnávají jednotlivé oblasti obou jazyků. Pátá kapitola se rovněž věnuje praktické části, a to v porovnávání složitosti jednotlivých jazyků na základě statické analýzy zdrojových kódů mobilní aplikace. Poslední kapitola poté obsahuje srovnání programátorské přívětivosti jazyka Swift a Objective-C, na základě osobních pocitů a zkušeností s vývojem pro mobilní platformy.

2 Vývoj jazyka Swift

Jazyk Swift byl představen společností Apple v roce 2014 na vývojářské konferenci WWDC. Apple tento jazyk uvedl jako novou alternativu k stávajícímu jazyku Objective-C, který již začínal postrádat moderní jazykové konstrukce. Klíčové vlastnosti nového programovacího jazyka Swift měly být hlavně bezpečnost, rychlost a moderní programátorské přístupy. Další výhodou měla být zpětná kompatibilita se všemi stávajícími knihovnami a také s jazykem Objective-C. Swift obdržel rovněž podporu správy paměti ARC ihned ve své první verzi, což přivítali všichni vývojáři pro platformy od společnosti Apple. Podle indexu společnosti TIOBE roste popularita jazyka Swift mezi vývojářskou komunitou meziročně v řádu desetin procent. Tento růst je určitě dán také tím, že procentuální podíl programovacího jazyka Objective-C klesá. To je způsobeno odlivem vývojářů, kteří přecházejí na nový jazyk Swift. V současné době se nachází programovací jazyk Swift na dvanácté pozici s 2.287 procentním zastoupením[5]. Tabulka číslo 1 znázorňuje stav indexu firmy TOIBE pro nejpoužívanější programovací jazyky.

Tabulka 1: TOIBE index nejpopulárnějších programovacích jazyků pro duben 2017

A	Programovací jazyk	Zastoupení v procentech
1	Java	15.568
2	C	6.966
3	C++	4.554
4	C#	3.579
5	Python	3.457
..	.....	.....
12	Swift	2.287
14	Objective-C	2.163

2.1 Swift 1

Na první verzi jazyka Swift začala společnost Apple pracovat již v roce 2010, pod vedením Chrise Lattnera. Swift se snažil oproti staršímu Objective-C implementovat nové přístupy z jiných programovacích jazyků, jako je například Haskell nebo Python a zároveň si ponechat zpětnou kompatibilitu s Objective-C. Náhodou není ani to, že Swift pro kompilaci zdrojových kódů používá LLVM kompilátor, který je dílem již výše zmíněného Chrise Lattnera. Mezi základní stavební kameny první verze tohoto jazyka patří:

- Kontrola přístupu
- Volitelné hodnoty a jejich řetězení
- Nové datové typy

- Přetěžování operátorů
- Protokoly
- Anonymní funkce
- Definice tříd v jednom souboru
- Entice

Po vydání první verze společnost pracovala na další aktualizaci jazyka, která přinesla nová vylepšení. Tato aktualizace nesla označení 1.1 a byla uvolněna 20. října 2014 spolu s vývojovým prostředím Xcode 6.1. Hlavním vylepšením bylo přidání Failable inicializátoru. Ten měl za cíl upozornit vývojáře na to, že daná třída, která tento inicializátor implementovala, mohla při vytváření nové instance vrátit hodnotu nil.

Poslední vylepšení, které Apple vydal pro první verzi jazyka Swift, bylo označeno číslem 1.2. Oficiální vydání proběhlo 9. února 2015 rovněž s novou verzí vývojového prostředí Xcode 6.3 beta. Tato vylepšení lze rozdělit na dvě části. V první části je cíleno na přidávání nových funkcionalit jazyka, jako je například vynucené přetypování nebo přidání nativního datového typu pro množinu. Druhá část je pak převážně cílena na optimalizaci a kompilaci jazyka Swift. U druhé části je dobré zmínit následující vylepšení:

- Inkrementální sestavování
- Rychlejší spouštění debug a release verzí
- Lepší diagnostika při kompilaci

2.2 Swift 2

Druhá verze na sebe nenechala dlouho čekat a firma Apple ji představila 8. června 2015, na své WWDC konferenci věnované vývojářům a technickým novinkám. Největší pozdvižení mezi vývojáři vyvolala zmínka o tom, že v některé z dalších verzí, bude Swift uvolněn jako otevřený software. Toto pozdvižení je odůvodněno hlavně tím, že Apple jen velmi zřídka sdílí své projekty s veřejností takto otevřeným způsobem. Společně s druhou verzí jazyka, došlo k vydání nové verze vývojového prostředí Xcode 7.0. Mimo samotné oznámení o tomto vydání, došlo také k výrazným změnám ve funkcionalitě jazyka. Mezi nejhlavnější nově přidané funkcionality patří:

- Ošetření výjimek
- Podmínky pro verze prostředí
- Nové klíčové slovo guard
- Enumerátor pro libovolný datový typ

Stejně jako u první, tak i u druhé verze, docházelo k vydávání nových vylepšení a aktualizací. Tou první aktualizací je nová verze s označením 2.1. Tato aktualizace byla uvolněna spolu s prostředím Xcode 7.1 beta 2. Mimo oprav některých chyb jazyka, stojí zmínit, že došlo také k přidání podpory řetězců v rámci interpolace.

Dne 3. prosince 2015 Apple dostál svého slova a uvolnil jazyk Swift, jako otevřený software. Při této příležitosti, tak poskytl veškeré zdrojové kódy na veřejném Git repositáři[1]. Dále Apple vytvořil a spustil novou webovou stránku[2], která obsahuje všechny potřebné informace o jazyku Swift. V neposlední řade bylo přínosem také to, že došlo k vydání podpory Swiftu i pro linuxové operační systémy.

Druhá verze jazyka Swift obdržela poslední aktualizaci 21. března 2016. Ta dostala číselné označení 2.2. I vrámci této aktualizace, došlo k uvolnění nové verze vývojového prostředí Xcode 7.3. Stejně jako v ostatních případech, tak i zde Apple opět přidal nová vylepšení a funkcionality pro programovací jazyk Swift. K novinkám, které tato aktualizace přinesla patří následující:

- Možnost použití klíčových slov v názvech
- Operátory pro entice
- Nové klíčové slovo associatedtype
- Výraz pro používání selektorů
- Pojmenovávání argumentů ve funkcích

Kromě přidání nových funkcionalit, se Apple rozhodl i jednu z funkcionalit přesunout do zastaralých. Tato změna vyvolala lehké pozdvižení, ale nakonec ji vývojářská komunita přijala. Konkrétně se jednalo o prefix a postfix operátory pro inkrementaci a dekrementaci čísla o jedničku. Společnost Apple toto odstranění komentovala tím, že způsobovala mezi novými programátory nejistotu v tom, jak se tyto operátory vyhodnocovaly.

2.3 Swift 3

Současná verze jazyka Swift, je označená verzí 3 a byla oficiálně uvedena 13. září 2016. Jedná se o doposud největší aktualizaci jazyka, která do této doby proběhla. Přibylo zde plno nových konstrukcí a změn v dosavadně používaných a zaběhlých funkcionalitách. S verzí 3, vyšlo také nové vývojářské prostředí Xcode 8.0, které bylo rovněž obohaceno o spoustu užitečných novinek. Následující seznam popisuje nejvýznamnější změny v této verzi:

1. Nové nativní datové typy Date, IndexPath, Notification a jiné
2. Úpravy jmenných konvencí ve stávajících knihovnách a rámcích

[1] Git repositář se zdrojovými kódy programovacího jazyka Swift, je dostupný na této adrese https://github.com/apple/swift

[2] Webovou stránku se všemi důležitými informacemi o jazyku Swift, lze nalézt na adrese https://swift.org

3. Přepracování původního GCD API do nové nativní podoby
4. Odstranění zápisu for cyklu tak jak jej známe z jazyka C
5. Importování datového typu id z Objective-C jako Any
6. Přidání nového Package Manageru, který se stará o správu závislostí

V době psaní této diplomové práce, došlo k uvolnění nové aktualizace s označením 3.1. Aktualizace byla firmou Apple uvolněna 27. března 2017 a jedná se tak o nejaktuálnější verzi jazyka Swift. Podobně jako u ostatních aktualizací i zde byly přidány nové vlastnosti. Součástí této aktualizace jsou opravy chyb, které byly způsobeny při běhu na operačních systémech Linuxu. Dále pak byly přidány Failable inicializátory pro všechny číselné datové typy. Mimo jiné také došlo k podpoře generických typů pro vnořené třídy. V neposlední řadě byly přidány nové metody *prefix* a *drop* pro *Sequence* protokol. Při zpracovávání informací pro tuto kapitolu, bylo vycházeno z oficiálních zdrojů [3] [4] společnosti Apple.

3 Interoperabilita jazyků

Při představení jazyka Swift se v prezentaci Applu objevila zmínka o tom, že nový jazyk bude zpětně kompatibilní s Objective-C. Díky tomu mohli vývojáři ve svých stávajících aplikacích používat jazyk Swift, spolu se staršími zdrojovými kódy, napsanými v Objective-C. Zpětná kompatibilita také zaručila použitelnost stávajících knihoven a rámců společnosti Apple, při psaní nových aplikací v jazyce Swift. Aby toho nebylo málo, společnost Apple umožnila použitelnost zdrojových kódu napsaných v jazyce Swift i zpětně v programovacím jazyce Objective-C. Díky tomu, bylo potřeba převést určitou funkcionalitu z jazyka Swift, zpět do Objective-C. Bohužel ne všechny nové funkcionality šlo převést, a tak zde panují jistá omezení, která nastávají při převodech mezi jazyky. Mezi nejkritičtější funkcionality Swiftu, které nelze převést do jazyka Objective-C jsou:

- Generické typy
- Entice
- Enumerátory vycházející z jiného datového typu než Int
- Vnořené typy
- Struktury

Aby bylo možné použít starší zdrojový kód napsaný v Objective-C i v jazyce Swift, je potřeba vytvořit speciálně určený soubor. Tento soubor nese označení Bridging Header a slouží ke specifikaci hlaviček tříd jazyka Objective-C, které mají být propagovány do jazyka Swift. Pokud chceme v opačném případě importovat zdrojové kódy jazyka Swift do Objective-C, musíme v hlavičkovém či zdrojovém souboru specifikovat importování *Swift.h* hlavičky. Tato hlavička je automaticky generována a nemusíme tak vytvářet žádný soubor navíc, jak tomu bylo v případě Bridging Headeru.

3.1 Propagace metod a vlastností

Při propagaci a mixování obou jazyků, dochází ke změnám mezi chováním funkcí a vlastností tříd. Tyto změny jsou hlavně způsobeny odlišností jazyků a také programátorskými konvencemi, které se u obou jazyků používají. Při deklaraci funkcí v Objective-C se používá konvence, kdy za názvem funkce následuje slovo *With* a za ním pak následují jednotlivé názvy parametrů. Ve Swiftu tato konvence neplatí, jelikož deklarace funkcí používají podobnou konvenci jako v jazyce C a C++. Z tohoto důvodu, dochází při propagaci k přejmenovávání těchto funkcí.

V ukázce zdrojového kódu číslo 1, lze vidět, jak přesně k přejmenovávání funkce z Objective-C dochází. Nejprve je odebrána celá část začínající slovem *With*, která je odebrána spolu s názvem prvního argumentu. Následně je slovo *With* použito jako název prvního argumentu, při volání funkce *updateValue*.

```
// Definition of updateValueWithString function in Objective-C
- (void)updateValueWithString:(NSString *)string {
    value = string;
}

// Usage of updateValueWithString function in Swift
object.updateValue(with: "Some value")
```

Výpis 1: Zdrojový kód demonstrující přejmenovávání funkcí

Podobně jako u deklarace funkcí, tak i u vlastností tříd, dochází k určitým úpravám při propagaci do jazyka Swift. Programovací jazyk Objective-C pro deklarování třídních vlastností, používá klíčové slovo *@property*. Za tímto klíčovým slovem následuje výčet volitelných značek, které blíže specifikují chování dané vlastnosti. Značky ovlivňující chování vlastností, musí být vhodně transformovány do jazyka Swift z toho důvodu, že velkou část z nich nepodporuje. Značky, u kterých dochází k transformaci jsou:

- readonly - je převáděna do jazyka Swift jako vypočtená vlastnost
- atomic, nonatomic - značky nejsou do jazyka Swift převáděny, ale zachovávají si své chování
- getter=, setter= - při transformaci těchto značek dochází k jejich ignoraci a do jazyka Swift se neprojevují
- assign, copy, strong, unsafe_unretained - stejně jako u značek atomic a nonatomic i zde nejsou značky převáděny, ale i přes to si zachovávají svou funkcionalitu
- nonnull, nullable, null_resettable - jsou do jazyka Swift převáděny jako volitelné a nenulové vlastnosti
- class, weak - tyto značky mají stejné chovaní jako v jazyce Swift, z tohoto důvodu nedochází k jejich transformaci

3.2 Přemostěné typy

Datové typy jsou nedílnou součástí programovacích jazyků. Ne jinak je tomu i v případě Objective-C nebo Swiftu. Každý z jazyků má své nativní datové typy. Z tohoto důvodu, je potřeba při interoperabilitě obou jazyků zajistit jejich přemostění. Například jazyk Objective-C obsahuje primitivní datové typy, jako jsou *int*, *double* či *BOOL*. Kdežto Swift primitivní datové typy vůbec neobsahuje. Podobný problém nastává i u ostatních datových typů.

Jazyk Objective-C neobsahuje struktury. Díky tomu nastává problém, protože Swift implementuje některé své datové typy jako struktury. Proto při transformaci datových typů mezi

jazyky dochází k převedení instancí struktur na nové instance tříd. Zdárným důkazem jsou například pole či řetězce, které jsou z datového typu *Array* a *String* převedeny na *NSArray* a *NSString*. Tento převod demonstruje níže uvedený zdrojový kód s číselným označením 2.

```
// Objective-C
@property NSArray<NSString *> *strings;
@property int number;

// Swift
var strings: [String]
var number: Int32
```

Výpis 2: Zdrojový kód, který demonstruje přemosťování datových typů

Apple má snahu na tom, implementovat co nejvíce datových typů nativně ve Swiftu. Nejvíce těchto typů přibylo ve verzi 3.0. Takto přidávané typy nebývají jen věrnou kopií z jazyka Objective-C. Snaží se tak přinést výhody plynoucí z nových konstrukcí, které jazyk Swift nabízí.

3.3 Chyby a výjimky

Ve verzi 2.0 přibyla do Swiftu nová možnost ošetřování výjimek podobně, jako je tomu i u ostatních programovacích jazyků. Tato možnost se nachází i v Objective-C, ale přesto se příliš často nepoužívá. Místo zachycování výjimek používá Objective-C pomocnou třídu *NSError*. Metody, u kterých může potencionálně nastat problém, přebírají adresu na instanci této třídy v jednom ze svých parametrů. V případě, že v dané metodě došlo k problému, je na adresu uložen nový objekt třídy *NSError*, který obsahuje potřebné informace k identifikování příčiny.

I tento problém dokázali inženýři vyřešit. Při konverzi mezi jazyky tak dochází k transformaci třídy *NSError* na klíčové slovo *throws*. Aby bylo možné tuto konverzi provést, je nutné uvést datový typ *NSError* u posledního argumentu v deklaraci funkce. V případě, že některá z funkcí zároveň vrací hodnotu datového typu *BOOL*, je tento fakt ignorován a v jazyce Swift bude funkce navracet hodnotu typu *Void*. Podobně jako při propagování metod i zde dochází k odstraňování jmenných konvencí z jazyka Objective-C. Odstraněna je tak přípona *WithError* a *AndReturnError* z názvu příslušné funkce. V určitých případech k této transformaci z *NSError* na *throws* nedochází.

- Pokud je datový typ NSError uveden v delegátní metodě
- V případě, že argument funkce přebírá anonymní funkci s parametrem NSError

Automatickou změnu lze potlačit pomocí předem definovaného makra *NS_SWIFT_NOTHROW*. Toto makro se postará o to, že v jazyce Swift již nebude použito klíčové slovo *throws*, ale funkce bude přebírat jako argument datový typ *NSError*. Předešle popsanou problematiku prakticky ukazuje zdrojový kód číslo 3.

```objc
// Objective-C
- (BOOL)throwingWithError:(NSError **)error;
- (BOOL)throwingWithError:(NSError **)error NS_SWIFT_NOTHROW;
// Swift
func throwing() throws
func throwingWithError(_ error: NSErrorPointer) -> Bool
```

Výpis 3: Zdrojový kód ukazující převod mezi NSError a klíčovým slovem throws

3.4 Volitelné a nenulové hodnoty

Hlavní výhodou nového jazyka Swift je bezpečnost, kterou zaručují volitelné a nenulové hodnoty. Ty ve svém principu zaručují to, že při jejich používání, by mělo dojít k eliminaci problému s nečekanými *nil* hodnotami. Jenže jazyk Objective-C tuto funkcionalitu neobsahuje. Z toho důvodu je potřeba pomocí anotací, dát jazyku Swift vědět, zda daná hodnota může nebo nemůže obsahovat *nil*.

Pro řešení tohoto problému byly do jazyka Objective-C přidány nové značky a anotace. Pro vlastnosti byly přidány značky *nonnull*, *nullable* a *null_resettable*. Přičemž pro anotování slouží *__Nonnull* a *__Nullable*. Pokud pro danou položku není uvedena anotace či značka, dochází tak k implicitnímu rozbalení hodnoty.

Objective-C také umožňuje použití maker *NS_ASSUME_NONNULL_BEGIN* a *NS_ASSUME_NONNULL_END* pro větší bloky zdrojového kódu. V praxi to pak umožní automatické nastavení značek a anotací u deklarací funkcí a vlastností. Zdrojový kód číslo 4 ukazuje jak lze značky a anotace použít.

```objc
// Objective-C
@property (nullable) NSString *string;
@property void(^_Nullable block)(NSNumber *_Nonnull);

// Swift
var string: String?
var block: ((NSNumber) -> Void)?
```

Výpis 4: Zdrojový kód ukazující použití značek a anotací v praxi

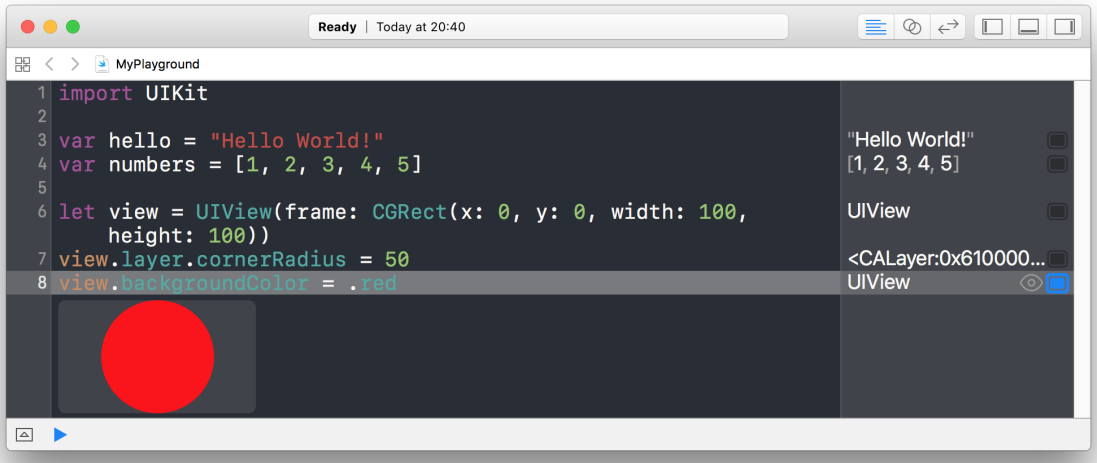
Při zpracovávání informací pro tuto kapitolu, bylo vycházeno z elektronické knihy [2], vydané společnosti Apple.

4 Vlastnosti jazyka Swift

Programovací jazyk Swift přebírá z jazyka Objective-C spoustu jeho výhod a mimo jiné využívá také stejné běhové prostředí. To umožňuje Swiftu běžet na stejných operačních systémech jako běží Objective-C. Pro připomenutí se jedná o následující systémy iOS, watchOS, tvOS a macOS. S příchodem uvolnění jazyka jako otevřený software, přišla i podpora pro linuxové systémy. To umožnilo použít tento programovací jazyk i mimo uzavřené platformy firmy Apple.

Díky společným rysům je možno ve Swiftu používat i knihovny a rámce napsané v Objective-C. Mezi ty nejpoužívanější patří Cocoa, Cocoa Touch a Foundtaion. To umožňuje jazyku Swift využít výhody obou jazyků. Oba jazyky rovněž používají stejné nástroje pro svou kompilaci. V našem případě se konkrétně jedná o LLVM a Clang. Díky tomu je možné využít technologie zvané bitcode. Ta se snaží o minimalizaci velikosti binárního souboru tím, že odstraňuje nepotřebný zkompilovaný zdrojový kód pro různé architektury. Tímto procesem lze podle Applu ušetřit až 50% z původní velikosti balíčku.

V dnešní době začíná pronikat programování i do základních škol. Toho si všimla i společnost Apple, a proto vložila velkou snahu vytvořit jazyk Swift tak, aby byl co nejvíce programátorsky přívětivý a dobře čitelný. Kvůli tomu vyvinul novou funkcionalitu pro vývojové prostředí Xcode, zvanou hřiště. V hřišti může programátor vyzkoušet všechny síly jazyka Swift a zároveň jsou jednotlivé příkazy přehledně graficky vyobrazeny. Obrázek číslo 1 ukazuje, jak takové hřiště vypadá.



Obrázek 1: Hřiště z vývojového prostředí Xcode

4.1 Vlastnosti

V oblasti programovacích jazyků je vlastností myšlena instanční proměnná s jistou možností kontroly nad jejím stavem. Ve výsledku to znamená, že můžeme ovlivňovat chování pří získávání čí nastavování hodnot proměnné. V jazyce Swift můžeme vlastnosti používat v rámci tříd, struktur a enumerátorů. Vlastnosti lze rozdělit do dvou skupin, na vypočtené a uložené. Vypočtené vlastnosti jsou takové vlastnosti, které svou hodnotu přímo neukládají ale vypočítávají na základě definovaných podmínek. Druhý typ je běžnější případ, kdy je hodnota uložena přímo v dané vlastnosti. Tento typ vlastností lze použít pouze v rámci definice tříd či struktur.

Pro kontrolu událostí, které vedou ke změně či získání hodnoty z vlastnosti, slouží klíčová slova, která se píší do složených závorek při její definici. V případě, že chceme zamezit změnám hodnoty vlastnosti, použijeme při definici klíčové slovo *let* místo *var*. V opačném případě můžeme použít následující klíčová slova, která nám jazyk Swift poskytuje:

- get - řeší proces vedoucí k získání hodnoty z vlastnosti
- set - umožňuje definovat akci, která povede k nastavení nové hodnoty
- willSet - tato akce je vyvolána před akcí set a umožňuje ošetření ještě předtím, než bude hodnota nastavena
- didSet - v tomto případě je akce vyvolána až po procesu, který vedl k nastavení nové hodnoty

V rámci většiny objektově orientovaných jazyků se používají speciální identifikátory, které slouží k zapouzdření jednotlivých funkcionalit. Nejinak je tomu i v případě jazyka Swift, kdy lze pro zapouzdření možno použít následující identifikátory *private*, *public*, *fileprivate* a *internal*. V případě jazyka Objective-C je tento proces řešen jinak. A to pomocí specifikace vlastnosti, buď v hlavičkovém nebo zdrojovém souboru. To zajistí skrytí či vystavení dané vlastnosti v rámci modulu.

Pro specifikaci typové vlastnosti lze v jazyce použít dvě různá klíčová slova. V obou případech mají stejnou funkcionalitu a záleží tak na zvyku programátora, které z nich zrovna použije. Mezi tato klíčová slova patří slovo *class* a *static*. Takto vytvořená vlastnost, může v budoucnu sloužit kupříkladu k vytvoření návrhového vzoru Singleton. Z důvodu, že Swift zaručí bezpečnost v případě, kdy by k dané vlastnosti před její inicializací přistoupili dvě vlákna najednou.

Poslední důležitý identifikátor pro vlastnosti se nazývá *lazy*. Ten umožní lazy loading pro danou vlastnost. Takové chování je dobré použít v případech, kdy je potřeba uchovávat obsah souboru či jiný časově náročný artefakt. Kromě jiného umožňuje v rámci definice vlastnosti používat klíčové slovo *self*. Identifikátor zaručí, že v době přístupu k vlastnosti bude vlastní instance alokována a inicializována. Ukázka zdrojového kódu číslo 5 demonstruje výše popsané informace o chování vlastností v jazyce Swift.

```swift
// Static computed property
static var number: UInt32 {
    get {
        return arc4random()
    }
}

// Private stored property
private var name: String

// Lazy property
lazy var now: Date = {
    return Date()
}()
```

Výpis 5: Zdrojový kód, který demonstruje použití vlastností v jazyce Swift

4.2 Datové typy

Datové typy neodmyslitelně patří ke každému programovacímu jazyku. Jazyk Swift obsahuje dva hlavní datové typy, a to třídy a struktury. Hlavním důvodem, proč nepoužívá jen třídy nebo jen struktury, je v rozdílném způsobu manipulace. Při vytvoření nové instance třídy, dochází k uložení reference do příslušné proměnné. U struktur se nepředává reference, ale hodnota. Tento přístup může být problematický, například při manipulaci s poli či slovníky, které obsahují velký počet záznamů. K eliminaci tohoto chování můžeme využít datové typy z rámce Foundation, napsaném v jazyce Objective-C. I v tomto případě nám možnost spojit oba jazyky nabízí silný nástroj, jak řešit určité problémy s lehkostí.

V jazyce Swift neexistují primitivní datové typy. Pro jejich ekvivalenty jsou vytvořeny nativní datové typy, které přidávají užitečnou funkcionalitu navíc. Kupříkladu pro datový typ *double*, existuje datový typ *Double*. Podobně tomu je i u ostatních primitivních typů. Jazyk Swift rovněž v nových verzích postupně přidává nativní ekvivalenty pro datové typy napsané v Objective-C. Mezi nově přidanými je například datový typ *Calendar* či *TimeInterval*.

Výše zmíněné datové typy nejsou jediné, které lze v rámci jazyka Swift použít. Prvními z nich jsou entice. Entice jsou datový typ, který posloupně obsahuje a slučuje jiné datové typy do jednoho celku. Entice je vhodné použít vpřípadech, kdy chceme, aby metoda vracela více než jednu hodnotu. Použíti entice je v tomto případě daleko vhodnější než například vracet kolekci hodnot.

Mezi další datové typy se řadí enumerátory. Ty v jazyce Swift mohou vycházet z libovolného datového typu, který splňuje následující implementaci protokolů *RawRepresentable*, *Equatable*

a *ExpressibleBy*. Enumerátory mohou obsahovat i definice funkcí či vypočtené vlastnosti. To přidává enumerátorům nové možnosti, jak je využít.

4.2.1 Třídy

Třídy jsou základním kamenem objektově orientovaných jazyků. Podobně je tomu i v případě Swiftu. Velkou výhodou jazyka Swift, je možnost definovat třídu v jednom zdrojovém souboru. Není tak potřeba vytvářet zvlášť soubor pro rozhraní a pro implementaci jako je tomu v případě Objective-C.

Třída v jazyce Swift může dědit pouze z jedné třídy. To ale neplatí pro protokoly. Těch může implementovat libovolné množství podle potřeby programátora. Stejně jako v jiných jazycích i zde můžeme ve třídě definovat konstruktor i destruktor. Další funkcionalitu, kterou třídy představují, je možnost použití vnořených typů. To může posloužit například k vytvoření anonymní třídy.

Podobně jako je tomu u vlastností i u tříd lze specifikovat režim přístupu pomocí identifikátorů *private*, *public*, *internal* a *fileprivate*. Společnost Apple nezapomněla ani na možnost třídám specifikovat a používat generické typy. Jak taková jednoduchá třída může vypadat, znázorňuje zdrojový kód číslo 6.

```
public class ClassB<T>: ClassA, CustomStringConvertible {
    public var data: [T]!
    public var description: String = "I am ClassB"
    override init() {
        data = []
    }
    private func addValue(value: T) {
        data.append(value)
    }
    deinit {
        data = nil
    }
}
```

Výpis 6: Zdrojový kód, který znázorňuje používání tříd v jazyce Swift

Z výše uvedené ukázky lze na začátku vyčíst definici třídy s identifikátorem přístupu *public*. Následně pokračuje název třídy a specifikace použití generického datového typu. Za dvojtečkou pak následuje předek, ze kterého třída *ClassB* dědí. *CustomStringConvertible* je poté protokol, který musí třída *ClassB* implementovat.

Dále jsou ve třídě specifikovány dvě vlastnosti. Ta první s názvem data, je datového typu pole, které obsahuje prvky generického typu *T*. Pak pokračuje definice vlastnosti *description*,

která je vynucená specifikací protokolu *CustomStringConvertible*. Jak si lze všimnout, tak u této vlastnosti dochází k nastavení výchozí hodnoty na řetězec „I am ClassB“.

Další definice již připadá pro konstruktor. Ten bývá označen klíčovým slovem *init*. V těle konstruktoru inicializujeme hodnotu vlastnosti data na prázdné pole. Pod specifikací konstruktoru se nachází privátní funkce, která přidává do pole novou hodnotu generického typu. Jako poslední je zde uveden destruktor, který při svém volání, nastaví hodnotu vlastnosti data na *nil*.

Stejně jako v Objective-C i v jazyce Swift lze třídy rozšířit o nové funkcionality. K tomu slouží klíčové slovo *extenstion*. Díky této technice můžeme rozšiřovat například třídy obsažené v knihovnách, bez nutnosti zásahu do jejich implementace. V rámci rozšíření lze definovat nové metody, vypočtené vlastnosti či přidat implementaci protokolu.

4.2.2 Struktury

V jazyce Swift mají struktury velmi podobné chování jako třídy. Ale najdou se zde rozdíly, které vedou k rozhodnutí, zda použít pro implementaci daného problému třídu či strukturu. Jeden z nejhlavnějších rozdílu je způsob manipulace. Tento problém byl popsán v předchozí podkapitole, věnující se právě třídám. Struktury, ale nabízejí i další omezení oproti třídám. Mezi tato omezení patří:

- dědění z jiné struktury či třídy
- definování destruktoru
- implementace třídních protokolů
- použití operátoru === pro porovnání referencí

Aby nebyly zmíněny jen nevýhody oproti třídám, je potřeba zmínit i jejich výhody. Struktury jsou při kompilaci daleko lépe optimalizovány než třídy. Tento fakt zapříčiňuje rychlejší vykonávání při běhu aplikace. Další výhodou je možnost specifikovat, zda bude instance struktury neměnná či naopak. Klíčové slovo *let* nám zaručí při vytváření nové instance to, že veškeré vlastnosti budou neměnné. V opačném případě použijeme klíčové slovo *var*. Pro někoho může být výhodou, že Swift automaticky generuje výchozí konstruktor z definovaných vlastností dané struktury. Jak takový proces vypadá, demonstruje ukázka zdrojového kódu číslo 7.

```
struct Structure {
    var number: Int
    let name: String = "Structure"
}

let instance = Structure(number: 1)
```

Výpis 7: Zdrojový kód, který uvádí praktickou implementaci struktury v jazyce Swift

4.3 Protokoly

Používání rozhraní je velmi často využívaný přístup v objektově orientovaných jazycích. Podobně je tomu i v případě Swiftu. Společnost Apple se ale rozhodla místo rozhraní používat protokoly. Protokoly jsou v zásadě velmi podobné rozhraním z jazyka Java nebo C#. Protokoly jsou v jazyce Swift velice silný nástroj a umožňují oproti rozhraním spoustu funkcionalit navíc.

První z funkcionalit je možnost deklarovat vlastnosti. U vlastností je možnost specifikovat, zda daná vlastnost bude dostupná pouze pro čtení či nikoliv. Dále lze pomocí klíčového slova *static* vynutit, aby se jednalo o typovou vlastnost. Kromě specifikování metod a vlastností, mohou protokoly definovat i konstruktory. Další užitečnou vlastností je možnost vynutit použitelnost protokolu pouze u tříd. Tohoto chování dosáhneme použitím klíčového slova *class*. Protože lze protokoly používat stejně jako ostatní datové typy, je zde možnost jejich řetězení. V praxi to znamená, že můžeme vynutit více protokolů pomocí operátoru &.

Podobně jako u tříd či struktur i zde můžeme protokoly rozšiřovat pomocí klíčového slova *extension*. Díky rozšíření můžeme implementovat výchozí chování pro protokolem deklarované metody a vlastnosti. Mimo jiné můžeme v rámci rozšíření definovat omezení. To zaručuje, že definované metody a vlastnosti nebudou dostupné, dokud třída či struktura neimplementuje protokol, který omezení vynutil. Ve zdrojovém kódu číslo 8 lze vidět, jak vypadá definice protokolu v praxi.

```
protocol Protocol {
    var number: Int { get set }
    func method()

    // Constructor
    init(number: Int)
}
```

Výpis 8: Zdrojový kód demonstrující použití protokolu

4.4 Volitelné a nenulové hodnoty

Již dříve bylo popsáno, že hlavním cílem jazyka Swift je bezpečnost. Není tím myšlena bezpečnost před zranitelnostmi, ale bezpečnost z hlediska psaní zdrojového kódu. Proto společnost Apple do jazyka Swift implementovala funkcionalitu volitelných hodnot. To znamená, že kromě specifikace datového typu určíme, zda může či nemůže nabývat hodnoty *nil*.

Pro tento účel slouží operátor *?*. Pokud tento operátor za datovým typem neurčíme, pak daná vlastnost či proměnná musí vždy obsahovat hodnotu. Kromě otazníku můžeme použít i operátor *!*. Jeho používání se ale nedoporučuje, a to z toho důvodu, že prakticky odstraňuje mechanismy bezpečnosti. V případě použití tohoto operátoru, dojde k tomu, že daná hodnota

může být nastavena na *nil*. Navenek se ale vlastnost nebo proměnná chová tak, že hodnotu *nil* obsahovat nemůže.

Pokud definujeme třídu nebo strukturu, která obsahuje vlastnosti bez operátorů *?* či *!*, je nutné implementovat konstruktor, ve kterém dojde k nastavení hodnot daných vlastností. Nebo nastavíme výchozí hodnoty při jejich definici. V případě, že některý z těchto operátorů použijeme, nastaví se tak výchozí hodnota na *nil*.

Další výhodou volitelných hodnot je řetězení. Toto chování přidává další míru bezpečnosti při psaní zdrojového kódu. Řetězení lze použít v případě volání metod, nebo při přístupu k jednotlivým vlastnostem. To zamezí volání metod na instance, které nejsou inicializované a přístup k hodnotám, které obsahují *nil*. Jak řetězení vypadá, můžeme vidět v ukázce zdrojového kódu číslo 9.

```
// Creation instance of optional Class type
let object: Class? = Class()

// Accessing optional class property
object?.number?.hashValue

// Invoking method on optional instance
object?.method()?.characters
```

Výpis 9: Zdrojový kód demonstrující řetězení

V případě, že pracujeme s volitelnými hodnotami, je potřeba získat reálnou hodnotu proměnné či vlastnosti. K tomu slouží dvě speciálně určené jazykové konstrukce. Tou první je rozbalování volitelných hodnot pomocí klíčového slova *if*. Druhou možností je použití klíčového slova *guard*, který podobně jako *if* umí danou hodnotu rozbalit. Jediný rozdíl je v řízení toku. Zatímco *if* v případě úspěšného rozbalení hodnoty pokračuje do true větve, klíčové slovo *guard* při selhání ukončí proces vykonávání v daném bloku kódu. Proces rozbalování volitelných hodnot pomocí *if* a *guard*, demonstruje zdrojový kód číslo 10.

```
// Unwraping optional value using if statement
if var number = number {
    number += 1
}

// Unwraping optional value using guard statement
guard var number = number else { return }
number += 1
```

Výpis 10: Zdrojový kód demonstrující rozbalování volitelných hodnot

K volitelným hodnotám se váže ještě jeden operátor. Tentokrát jde o binární operátor *??*. Ten slouží rovněž k rozbalení volitelné hodnoty. V případě že rozbalení selže, je nastavena výchozí hodnota z pravé strany operátoru.

Operátor jednoho otazníku lze rovněž použít k přetypování. V takovémto případě je otazník uveden za klíčovým slovem *as*. Pokud přetypování selže, nezpůsobí to pád programu. Místo toho bude příslušná hodnota nastavena na *nil*.

4.5 Anonymní metody

Inženýři společnosti Apple se rovněž rozhodli do jazyka Swift přidat podporu anonymních metod. Anonymní metody hrají v moderních programovacích jazycích důležitou roli. Díky jejich používání šetří čas a také přidávají silný nástroj, jak řešit některé problémy přímočaře.

Anonymní metody lze vytvářet čistě uvnitř nějakého bloku kódu, pro lepší rozčlenění složitější funkcionality. Dále je lze ukládat do proměnných a následně je vykonávat. V neposlední řadě je lze použít v argumentech funkcí.

Tyto metody mohou rovněž obsahovat vlastní argumenty a návratovou hodnotu. Při deklaraci argumentů můžeme, ale nemusíme udávat jejich název. Pokud není název specifikován, je poté název pro argument vytvořen při definici anonymní metody. Pokud chceme anonymní metody použít pro jednoduché operace, je možné využít operátor *$*. Ten následně od indexu 0 až po počet argumentů anonymní metody automaticky vybere daný argument.

Při vytváření anonymní metody, dochází k zajištění všech referencí na proměnné, které jsou v daném kontextu použity. Pokud použijeme anonymní metodu jako argument ve funkci, musíme brát v potaz chování označené escaping a noescaping. V případě, že daná anonymní metoda bude vykonávaná asynchronně, je nutné specifikovat před daným argumentem klíčové slovo *@escaping*. Tím se zaručí, že daná anonymní metoda může opustit vykonávání dané funkce. Ve výchozím stavu jazyk Swift automaticky před každý argument s anonymní metodou přidává *@noescaping*. Na základě těchto chování může dojít k příslušné optimalizaci a zrychlení běhu programu. Ve zdrojovém kódu číslo 11, lze vidět příklad anonymních metod v jazyce Swift.

```swift
// Anonymous method stored in variable
var closure = { print("Print something") }

// Definition of method with anonymnous method argument
func method(closure: (Bool) -> ()) {
    closure(true)
}

// Usage of shorthand arguments
[1, 2, 3].sorted(by: { $0 > $1 })
```

Výpis 11: Zdrojový kód popisující použití anonymních metod

4.6 Operátory

Operátory v jazyce Swift jsou prakticky srovnatelné s operátory z ostatních programovacích jazyků. Ve Swiftu tak najdeme unární, binární a ternární operátory. K použití je zde sada běžných operátorů pro přiřazení, porovnání, aritmetické operace, logické operace a bitové operace.

Nalezneme zde i netradiční operátory. Prvním z nich je operátor rozsahu označený třemi tečkami. Ten má své nejběžnější použití například v cyklech. Podstata tohoto operátoru je vygenerovat posloupnost celých čísel v určitém rozsahu, kdy definujeme spodní a horní hranici. Dalším z nich je operátor *??*, který byl popsán v podkapitole volitelných a nenulových hodnot.

Přetečení a podtečení jsou známé pojmy, které můžou způsobit nechtěné problémy. V jazyce Swift, jsou tato chování ve výchozí podobě zakázána. Místo toho je agresivně vyhozena chyba, která ukončí běh programu. V takovém případě lze lépe odchytit, proč daný výpočetní úkon nepracuje korektně. V případě, že chceme přetečení a podtečení vynutit, je potřeba před aritmetickým operátorem uvést &.

Nově lze také operátory přetěžovat. K přetěžování lze použít všechny typy operátorů, kromě operátoru přiřazení a operátoru ternární podmínky. Tento přístup například používá protokol *Equatable*. Ten deklaruje nutnost implementace přetíženého operátoru pro porovnávání. Při přetěžování, je potřeba také myslet na prioritu. Tu lze nastavit pro každý operátor zvlášť. Jak takový přetížený operátor vypadá, demonstruje zdrojový kód číslo 12.

```
// Declaration of overloaded operator ==== with ComparisonPrecedence priority
infix operator ==== : ComparisonPrecedence

// Behavior definition of overloaded operator
func ====(lhs: String, rhs: String) -> Bool {
    return lhs.characters.count == rhs.characters.count
}
```

Výpis 12: Zdrojový kód popisující použití anonymních metod

4.7 Správa paměti

Toto ožehavé téma je problematické pro všechny nové programátory. Ještě donedávna musel programátor v jazyce Objective-C znát všechny životní cykly instancí objektů a proměnných. Aby toho nebylo málo, musel spravovat destrukci nepotřebných instancí. To se ale firma Apple rozhodla změnit a uvedla technologii s názvem ARC. Ta značně zjednodušila správu paměti v jazyce Objective-C a ušetřila tak hodně vrásek na čele mnoha programátorů.

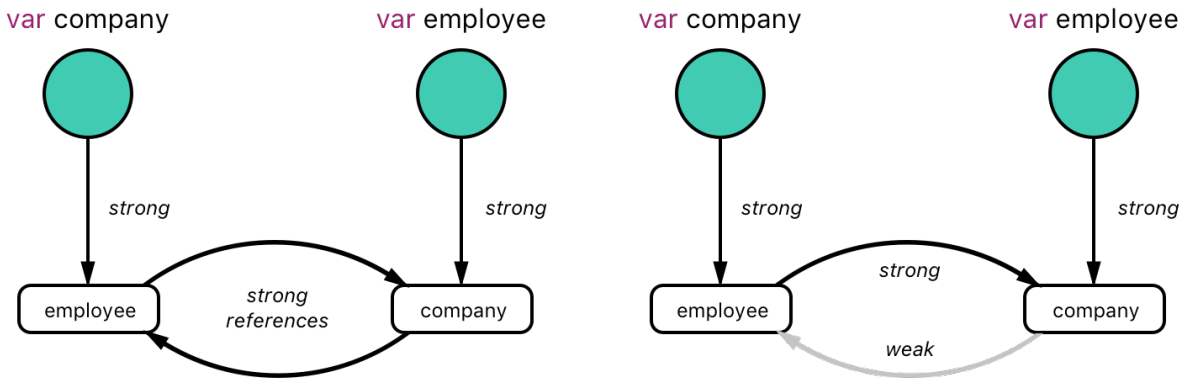
ARC zajišťuje správu paměti v rámci celého programu. Při vytváření nové instance třídy se automaticky alokuje blok v paměti. Ten vyhrazuje prostor pro všechny uložené vlastnosti dané třídy a také informace o typu instance.

V rámci jazyka Swift rozlišujeme dva typy referencí. Prvním typem je reference *strong*. Ta je v případě definování vlastností třídy nastavena automaticky. Její chování vyplývá z jejího názvu. Drží tak silnou referenci na danou instanci. To znamená, že do doby, dokud bude silná reference na danou instanci existovat, nebude odkazovaná instance dealokována. Pokud je v opačném případě specifikována daná reference jako *weak*, vytvoří se tak slabá reference. V takové situaci je tato vazba při dealokaci ignorována.

Tyto dva přístupy jsou pro práci ARC stěžejní. V praxi tato technologie funguje tak, že u každé instance počítá všechny odkazující silné reference. Při vytvoření nové silné reference na danou instanci se navýší čítač o jedničku. V případě, že je některá ze silných referencí odstraněna, je čítač snížen o jedničku. Jakmile je čítač referencí roven nule, je daná instance ihned dealokována. Podobné chování mají i garbage collectory v jiných jazycích. Ty ale nedealokují instance ihned, ale čekají, než se zvýší požadavek na paměť nebo je promazávají v intervalech.

Jazyk Swift nabízí i třetí typ reference. Tou je *unowned* reference. Ta má prakticky stejné chování jako *weak* reference. Její použití má smysl, pokud je známo, že životnost dané instance bude trvat stejně dlouho, jako životnost vlastníka. Díky tomu není potřeba danou vlastnost specifikovat jako volitelnou. To umožní lepší optimalizaci programu.

Kvůli tomuto chování technologie ARC, je potřeba při definování tříd myslet na možné cyklické vazby. Ty mohou způsobit problém, při kterém budou dvě instance udržovat silnou referenci do kříže. Tento problém způsobí, že nikdy nedojde ke snížení čítače na nulu a vznikne tak memory leak. Proto je při těchto situacích dobře promyslet, zda použít silnou nebo slabou referenci. Obrázek číslo 2 demonstruje, jak cyklická vazba vypadá a jak ji lze vyřešit pomocí *weak* reference.



Obrázek 2: Ukázka cyklické vazby a jejího řešení

Při implementaci anonymních metod rovněž dochází k vytváření silných referencí. Proto je potřeba dávat si pozor i vtěchto situacích. Pro vytvoření slabých referencí, je potřeba použití spe-

ciální syntaxe. Ta specifikuje, na jaké proměnné a vlastnosti budou v daném kontextu vytvořeny slabé reference. Tuto syntaxi demonstruje ukázka zdrojového kódu číslo 13.

```
var string = NSMutableString()

// Anonymous method with weak reference to NSMutableString instance
let closure = { [weak string] in
    string?.append("I am weak :(")
}
```

Výpis 13: Zdrojový kód, který demonstruje slabou referenci u anonymní metody

4.8 Správce závislostí

Správa závislostí byla donedávna realizována skrze aplikace třetích stran. Mezi ty nejpoužívanější patří například Cocoapods[3] či Carthage[4]. Třetí verze jazyka Swift vše změnila a přinesla novinku v podobě integrovaného správce závislostí.

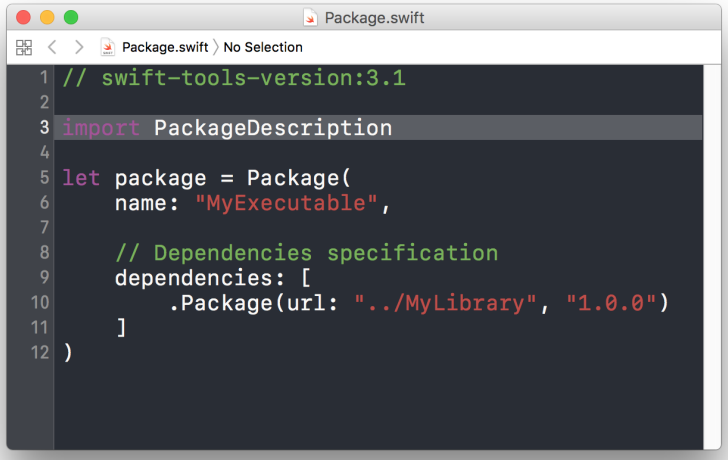
Pro vytváření a správu závislostí se používá příkazová řádka. To je ovlivněno tím, že vývojové prostředí Xcode vsoučasné době neintegruje správce závislostí. Pro vytvoření nové knihovny se správou závislostí je potřeba použít příkaz *swift package init –type library*. Tím dojde k vytvoření výchozího projektu, který obsahuje v kořeni složky soubor *Package.swift*. Ten obsahuje informace o knihovně a specifikace dalších závislostí. Dále jsou vytvořeny dvě složky s názvem *Sources* a *Tests*. Ty obsahují soubory se zdrojovými kódy a testy.

Aby šlo výslednou knihovnu použít jako novou závislost v jiném projektu, je potřeba v kořeni složky inicializovat nový git repositář. Po inicializaci repositáře potvrdíme všechny změny a vytvoříme značku pro novou verzi. Cesta k repositáři a číslo verze je později použita ke specifikaci závislosti.

Pokud chceme výslednou knihovnu použít v projektu, je potřeba opět využít příkazovou řádku. Do ní je nutné vepsat příkaz *swift package init –type executable*. Ten vytvoří výchozí projekt, ale nyní ne pro knihovnu, nýbrž pro spustitelný program. V kořeni se opět nachází soubor *Package.swift*, který obsahuje stejné informace jako v předchozím případě. Podobně se vytvoří i složky *Sources* a *Tests*. Nyní ale složka *Sources* obsahuje soubor *main.swift*, který implementuje zdrojový kód pro počáteční bod programu.

Pro přidání nové závislosti do spustitelného projektu je potřeba upravit soubor *Package.swift*. Do něj je nutné specifikovat závislost na knihovnu, která byla vytvořena v minulosti. K tomu je třeba znát cestu ke git repositáři dané knihovny a ideálně i její verzi. Jak může výsledná podoba souboru *Package.swift* vypadat, je možné vidět na obrázku číslo 3.

[3] Webové stránky projektu Cocoapods se nacházejí na adrese https://cocoapods.org/
[4] Git repositář projektu Carthage je dostupný na této adrese https://github.com/apple/swift



Obrázek 3: Příklad vzhledu souboru pro správu závislostí

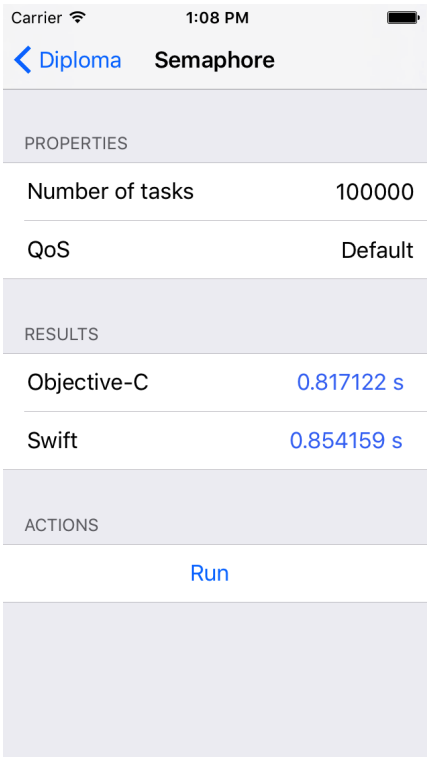
Při zpracovávání informací pro tuto kapitolu, bylo vycházeno z elektronické knihy [1], vydané společnosti Apple.

5 Praktické porovnání rychlosti

Pro porovnání rychlosti byla vytvořena mobilní aplikace, která má otestovat rozdíly mezi jazykem Swift a Objective-C. Aplikace je rozdělena do tří logických celků. Prvním z nich je knihovna s názvem *obj*. Ta obsahuje všechny zdrojové kódy implementované v jazyce Objective-C. Druhým logickým celkem je knihovna *swt*. Která pro změnu obsahuje všechny zdrojové kódy napsané v jazyce Swift. Posledním celkem je výchozí projekt s názvem *diploma*. Ten integruje obě zmíněné knihovny. Mimo jiné obsahuje uživatelské prostředí, ve kterém může uživatel spouštět jednotlivé testy. Při implementaci všech testů byla vytvořena verze v jazyce Swift a verze v jazyce Objective-C. Obě tyto verze využívali výhod vybraného jazyka.

Kromě výše zmíněných knihoven obsahuje výchozí projekt implementaci struktury *TimeMeasurement*. Struktura má za úkol měření rychlosti jednotlivých úloh. Jednotlivá měření se pouštějí v novém vlákně. Toto chování je odůvodněno tím, aby nedocházelo k zamrzání uživatelského prostředí během samotného testování. Jednotlivé časy měření jsou následně navráceny v argumentu anonymní metody *completion*.

Výsledná aplikace tak obsahuje seznam položek pro jednotlivé testy. Po vybrání příslušného testu, je možné spustit jeho vykonávání. Před spuštěním testování, může uživatel nastavit výchozí hodnoty, které daný test ovlivňují v jeho časové náročnosti. Všechna provedená testování byla spuštěna na iPhonu 6S s operačním systémem iOS ve verzi 10.3.1. Na uvedeném obrázku číslo 4 lze vidět, jak vypadá výsledný test v mobilní aplikaci.



Obrázek 4: Ukázka testu z výsledné mobilní aplikace

Všechny výsledky testů jsou uloženy v příloze na CD. Výsledky jsou strukturovány v Excel tabulce. U všech testů jsou uvedeny zvolené parametry pro měření. Měření byla spuštěna alespoň pětkrát proto, aby se eliminovala případná chyba měření.

5.1 Manipulace s kolekcemi

Jednotlivé testy jsou rozděleny do určitých celků podle jejich zaměření. První z nich se zaobírá manipulacemi s kolekcemi. Mezi tři použité kolekce se řadí pole, množina a slovník. U všech těchto tří kolekcí se testovala rychlost následujících funkcionalit.

- Inicializace
- Třídění
- Filtrování
- Přístup k prvkům
- Mutace

V případě inicializace dochází k vytváření nových prázdných kolekcí. V dalším případě se testuje rychlost, s jakou dokáže daná kolekce setřídit všechny její prvky. U filtrování dochází k vyhledání a vybrání jen určitých prvků z dané kolekce na základě jednoduché podmínky. Při přístupu k prvkům se testuje čas, za který se provede získání určitého počtu prvků z dané kolekce. Poslední test měří rychlost přidávání nového prvku do vybrané kolekce. U všech výše uvedených testů, je možnost v mobilní aplikaci nastavit počet prvků nebo kroků, se kterými daná kolekce bude operovat.

5.1.1 Pole

U testování polí byla v případě Swiftu použita nativní struktura *Array*. V případě Objective-C, pak třída *NSArray*. To umožnilo testovat výhody nativní implementace kolekce v jazyce Swift a Objective-C. Tabulka číslo 2 ukazuje výsledky měření pro oba jazyky.

Tabulka 2: Výsledky testování pro pole

Test	Objective-C	Swift
Inicializace	0,089 s	0,031 s
Třídění	0,512 s	0,450 s
Filtrování	0,126 s	0,177 s
Přístup k prvkům	0,056 s	0,001 s
Mutace	0,055 s	21,656 s

Z tabulky lze vyčíst, že je jazyk Swift v mnoha případech rychlejší nebo srovnatelně rychlý s jazykem Objective-C. Nejrazantnější rozdíl v časech nastává při mutacích kolekce. Zde jazyk Objective-C těží z použití třídy místo struktury. To umožňuje předávat pole odkazem, a ne hodnotou jako v případě Swiftu. Ten musí nejprve zkopírovat obsah celého pole a následně přidat prvek. Proto je dobré předem zvážit, zda v jazyce Swift použít nativní datový typ pro pole či využít třídu *NSArray* z jazyka Objective-C.

5.1.2 Slovník

Podobně jako u polí i u slovníku se testovaly výkonnostní rozdíly mezi nativní kolekcí *Dictionary* z jazyka Swift a *NSDictionary* z jazyka Objective-C. Při testování byly stejně jako v předchozím případě porovnávány jednotlivé funkcionality uvedené v této kapitole. Výsledky všech měření je možné vidět v uvedené tabulce číslo 3.

Tabulka 3: Výsledky testování pro slovník

Test	Objective-C	Swift
Inicializace	0,101 s	0,056 s
Třídění	1,171 s	0,883 s
Filtrování	1,790 s	0,001 s
Přístup k prvkům	0,189 s	0,076 s
Mutace	0,076 s	6,954 s

Výsledky měření jsou opět ve prospěch jazyka Swift. Jediný značný rozdíl v časech nástává opět při testování mutace kolekce. Kdy je v jazyce Swift datový typ *Dictionary* reprezentován strukturou. To dává opět výkonnostní výhodu třídě *NSDictionary* z jazyka Objective-C. Jelikož se nepředává hodnotou, nýbrž odkazem. Z toho důvodu je potřeba brát zřetel, kdy v jazyce Swift použít pro slovník nativní datový typ či využít třídní reprezentaci z jazyka Objective-C.

5.1.3 Množina

Poslední testovanou kolekcí je množina. Ta v případě Swiftu využívá nativně datový typ *Set*. Obdobně je tomu i v jazyce Objective-C, který pro tento účel používá třídu *NSSet*. Testování obou přístupů probíhalo obdobně jako v předešlých případech. Časy jednotlivých měření prezentuje tabulka číslo 4.

Tabulka 4: Výsledky testování pro množinu

Test	Objective-C	Swift
Inicializace	0,290 s	0,030 s
Třídění	0,801 s	0,452 s
Filtrování	0,222 s	0,182 s
Přístup k prvkům	0,115 s	0,034 s
Mutace	0,075 s	14,002 s

I v případě množin lze z výsledné tabulky vyčíst, že jazyk Swift předhání svého staršího soupeře. Opět se zde objevuje stejný propad výkonu při testování mutací. Z tohoto rozdílu je patrné, že kopírování všech hodnot a přidání nového prvku navíc, je v případě struktury *Set* zásadní problém. Proto stejně jako u polí či slovníku, je dobré dopředu promyslet, kdy bude lepší pro množinu využít nativní datový typ nebo třídní reprezentaci *NSSet* z jazyka Objective-C.

5.2 Práce s vlákny

Donedávna byla konstrukce pro práci s vlákny pro oba jazyky stejná. Změna přišla až ve třetí verzi jazyka Swift. V té přibyla nativní implementace pro správu vláken. Z toho důvodu je možné prakticky porovnat rychlost obou řešení.

V mobilní aplikaci je přímo vyhrazená sekce pro testování vláken. V ní se testuje vytváření nových vláken, práce se semafory a synchronizace úkolů pomocí skupin. Jednotlivé testy obsahují volbu pro prioritu, se kterou budou daná vlákna vytvořena a spravována.

První typ se jmenuje *User interactive* a používá se v případě, že chceme pracovat s uživatelským rozhraním. Kvůli tomu má tento typ nejvyšší prioritu ze všech. Dalším typem je *User-initiated*. Tento typ má o něco nižší prioritu a je vhodné jej použít v případě, že daný úkon chceme co nejrychleji zpracovat a zobrazit uživateli výsledek. Třetím typem je *Utility*. Tento typ má opět o něco nižší prioritu a je vhodné jej použít v případě, že chceme získat data ze vzdáleného zdroje. Posledním ze skupiny těch nejběžnějších typů je *Background*. Jeho priorita je ze všech typů nejnižší. Proto je vhodné ho využit v situacích, kdy je například potřeba vytvořit zálohu.

Tyto čtyři nejběžnější typy doplňují další dva. Jedná se o typ *Default* a *Unspecified*. První z těchto dvou typů spadá svou prioritou mezi *User-initiated* a *Utility*. Tento typ je nastaven jako výchozí v případě, že nedefinujeme žádný z předešle uvedených typů. Druhý typ *Unspecified* má rovněž specifické chování. Je-li tento typ nastaven, dojde k automatickému zvolení priority až za běhu programu.

5.2.1 Vytváření nových vláken

První z testů se věnuje vytváření nových vláken. Hlavním smyslem tohoto testování, je změřit rychlost jednotlivých řešení pro daný jazyk. V případě jazyka Swift, je pro vytváření nových vláken použita nativní třída *DispatchQueue*. V opačném případě, je pro jazyk Objective-C použita globální funkce s názvem *dispatch_async*. Naměřené hodnoty lze vyčíst z uvedené tabulky číslo 5.

Tabulka 5: Výsledky testování pro vytváření nových vláken

Typ priority	Objective-C	Swift
User interactive	0,423 s	0,730 s
User-initiated	0,404 s	0,572 s
Utility	0,403 s	0,549 s
Background	0,403 s	0,541 s

Z naměřených hodnot vychází, že implementace v jazyce Objective-C je o něco málo rychlejší nežli v jazyce Swift. Paradoxně největší rozdíl nastává u vláken vytvořených pro nejvyšší prioritu. V ostatních případech jde o rozdíl v rámci jedné desetiny sekundy.

5.2.2 Používání semaforu

Semafory se často používají v programovacích jazycích pro synchronizaci vláken. V jazyce Swift je pro tuto funkcionalitu vyhrazena třída *DispatchSemaphore*. V rámci této třídy, je řešeno i zamykání a odemykání semaforu. Obdobně je to řešeno i v rámci Objective-C. Ten pro reprezentaci semaforu využívá datový typ *dispatch_semaphore_t*. Tento datový typ však neřeší odemykání a zamykání semaforu. Pro tuto funkcionalitu je nutné využít globální metodu *dispatch_semaphore_signal* pro odemknutí a *dispatch_semaphore_wait* pro zamknutí.

Rychlosti obou použitých řešení jsou testovány v rámci mobilní aplikace. Výsledky testů lze vidět v uvedené tabulce s označením číslo 6.

Tabulka 6: Výsledky testování pro práci se semafory

Typ priority	Objective-C	Swift
User interactive	2,458 s	2,419 s
User-initiated	2,130 s	2,455 s
Utility	2,523 s	2,458 s
Background	2,562 s	2,505 s

Hodnoty z tabulky ukazují, že výsledky měření pro jednotlivá řešení vycházejí podobně. Nejvyšší rozdíl mezi časy, je nyní naměřen pro druhou nejvyšší prioritu. Ten činí pro jazyk Swift tři

desetiny sekundy. Pro ostatní priority jsou rozdíly zanedbatelné a nelze tak určit jednoznačného vítěze.

5.2.3 Synchronizace dílčích úkolů

Pro synchronizaci dílčích úkolů se v rámci jazyka Swift a Objective-C používá přiřazování jednotlivých úloh do skupin. V rámci jazyka Swift je použita třída *DispatchGroup*, která zastřešuje veškerou funkcionalitu. U jazyka Objective-C se o reprezentaci skupiny stará datový typ s názvem *dispatch_group_t*. Vstup a výstup ze skupiny řeší speciální globální funkce s názvem *dispatch_group_enter* a *dispatch_group_leave*.

Tato odlišná řešení byla v rámci mobilní aplikace otestována na základě jejich rychlostí. Časy jednotlivých měření pro dané řešení obsahuje tabulka číslo 7.

Tabulka 7: Výsledky testování pro synchronizaci dílčích úkolů

Typ priority	Objective-C	Swift
User interactive	3,133 s	3,096 s
User-initiated	2,943 s	2,956 s
Utility	2,195 s	2,152 s
Background	2,160 s	1,924 s

Naměřené časy opět dokazují, že v případě synchronizace dílčích úkolů nedochází k výraznému rozdílu. Největší rozdíl nastává u nejnižší priority. Kde rozdíl činí dvě desetiny sekundy ve prospěch jazyka Swift. Z toho lze usoudit, že jednotlivá řešení jsou rychlostně srovnatelná.

5.3 Přístup k hardwaru

Jazyk Swift a Objective-C se v dnešní době využívá k programování mobilních aplikací pro operační systémy Applu. Kvůli tomu musela být do obou jazyků přidána podpora knihoven, které umí s daným hardwarem mobilního zařízení pracovat. V rámci testování přístupu k hardwaru v mobilních zařízení, byly otestovány tyto funkcionality:

- práce s vnitřním úložištěm
- získávání dat z GPS modulu
- přístup k fotoaparátu

Všem těmto testům byla v mobilní aplikaci vyhrazena speciální sekce s názvem *Hardware*. U všech těchto testů je stejně jako v jiných testech nabízená možnost zvolení parametrů, které zvyšují časovou náročnost.

5.3.1 Zapisování a čtení do souborového systému

Tento test vnitřního úložiště se dělí na dvě části. První z nich obsahuje testování zápisu do souborového systému. V případě druhé části se testuje čtení ze souborového systému. Obě tyto části využívají v rámci jazyka Swift nativní třídu *FileManager*, která zajišťuje veškerou funkcionalitu pro práci se souborovým systémem. U jazyka Objective-C, je poté použita třída *NSFileManager*, která poskytuje stejnou funkcionalitu jako je tomu v případě jazyka Swift.

Při testování zápisu a čtení, lze nastavit kolik souborů bude zapsáno a načteno. Mimo to lze dále specifikovat i velikosti těchto souborů. Díky tomu je při testování zajištěna jistá flexibilita. V rámci testování nejprve došlo k otestování čtecího a následně zapisovacího procesu. Jednotlivé parametry jsou pro oba jazyky totožné. Výsledné časy zápisu a čtení pro oba jazyky reprezentuje uvedená tabulka s číslem 8.

Tabulka 8: Výsledky testování pro zápis a čtení ze souborového systému

Typ	Objective-C	Swift
Zápis	0,429 s	0,301 s
Čtení	0,014 s	0,012 s

Z výsledků je patrné, že nativní implementace v jazyce Swift dosahuje při zapisování souborů lehkého náskoku. Tento náskok činí něco málo přes jednu desetinu sekundy. V případě čtení je ale rozdíl natolik těsný, že nelze s jistotou určit jasného vítěze testu.

5.3.2 Získávání geolokace

V dnešní době si lze jen s těží představit mobilní zařízení, které by neobsahovalo GPS modul. Apple přidal podporu této technologie až v rámci iPhonu verze 3G. Proto i zde musel vytvořit novou knihovnu, která by uměla zpracovávat data z GPS modulu. Oba jazyky tak využívají pomocnou třídu *CLLocationManager* a *CLGeocoder* z knihovny *CoreLocation*. Tato knihovna zajišťuje veškerou potřebnou funkcionalitu k získání geolokace daného mobilního zařízení. Informace o lokaci poskytuje knihovna ve formě zeměpisné šířky a délky.

Testování se dělí na dvě roviny. V první z nich jsou získávána data o zeměpisné délce a šířce. Při získávání těchto dat, lze vybrat určitou přesnost na kterou je dána pozice určena. Toto nastavení poté zvyšuje čas potřebný k získání lokace mobilního zařízení. Mezi tato nastavení patří:

- BestForNavigation - nejvyšší možná přesnost určená pro navigaci
- Best - druhá nejvyšší možná přesnost v rámci tří metrů
- NearestTenMeters - přesnost v okruhu deseti metrů

- HundredMeters - čtvrtá nejvyšší přesnost v řádu sta metrů
- Kilometer - přesnost v okruhu jednoho kilometru
- ThreeKilometers - nejnižší možná přesnost v rámci tří kilometrů

Druhá rovina pak získává informace o adrese pro zadanou zeměpisnou šířku a délku. Při všech měřeních byla použita druhá nejvyšší možná přesnost. Naměřené hodnoty pro obě zmíněné roviny reprezentuje tabulka soznačením číslo 9.

Tabulka 9: Výsledky testování pro získávání geolokace

Typ	Objective-C	Swift
Lokace	10,004 s	10,006 s
Adresa	0,103 s	0,088 s

Z naměřených časů vyplývá, že rozdíly pro oba jazyky jsou naprosto minimální. Z tohoto důvodu nelze určit, který jazyk je na tom rychlostně lépe. Časy se neliší zřejmě proto, že se v obou jazycích využívají třídy ze stejné knihovny *CoreLocation*.

5.3.3 Práce s fotoaparátem

V rámci testování přístupu hardwaru, byla otestována i práce s fotoaparátem. Smyslem testu bylo, změřit čas potřebný k vyfocení jedné fotografie. Testován byl jak přední fotoaparát, tak zadní. Pro vyfocení nové fotografie byla použita pomocná třída z knihovny *AVFoundation*. Implementace této knihovny je pro jazyk Swift a Objective-C společná. Při testování bylo potřeba zajistit, aby testované zařízení setrvalo ve stejné poloze. Časy potřebné k vyfotografování nové fotografie demonstruje uvedená tabulka číslo 10.

Tabulka 10: Výsledky testování pro práci s fotoaparátem

Kamera	Objective-C	Swift
Zadní	0,859 s	0,711 s
Přední	0,560 s	0,571 s

Časové rozdíly mezi přední a zadní kamerou jsou způsobeny hlavně tím, že zadní kamera má daleko vyšší rozlišení. To znamená, že se zvyšují nároky na pořízení takovéto fotografie. Rozdíly časů mezi jazykem Swift a Objective-C jsou v případě použití přední kamery zanedbatelné a nelze tak určit, který jazyk je na tom rychlostně lépe. V případě zadní kamery lze ale vidět, určitý rozdíl v řádu jedné desetiny sekundy. Tento rozdíl sice není znatelný, ale i tak k němu dochází. Přestože je v obou jazycích využita stejná knihovna *AVFoundation*.

5.4 Zobrazování uživatelského prostředí

Při vytváření aplikací, je potřeba data prezentovat uživateli. Proto je důležité, aby odezva prostředí byla co nejrychlejší. Proto, aby uživatel nemusel čekat, než se určité prvky vykreslí. Při vývoji aplikací pro operační systémy Applu se pro reprezentaci uživatelského prostředí používá technologie zvaná *Storyboard*.

Jedná se o grafické prostředí, kde může programátor tvořit jednotlivé obrazovky. Na ty potom může vkládat jednotlivé komponenty, jako například popisky, textová pole či tlačítka. Obrazovky pak lze pomocí přechodů propojovat. Dříve se pro jednotlivé obrazovky musely vytvářet *xib* soubory. Ty reprezentovaly jednotlivé obrazovky, avšak přechody mezi nimi musely být řešeny pomocí implementace ve zdrojovém kódu.

Obě tyto technologie jsou dostupné jak v jazyce Objective-C, tak ve Swiftu. Z tohoto důvodu je možné jejich otestování v rámci vytvořené mobilní aplikace. Ta otestuje rychlost pro vykreslování objektů na plátno a dynamické přidávání nových prvků do listu.

5.4.1 Vykreslování objektů na plátno

První úloha, která je v rámci aplikace testována, je vykreslování objektů na plátno. V této úloze dochází k postupnému vykreslování jednotlivých obrazců na plátno. Jako objekty pro vykreslení posloužily kruhy, čtverce a kříže. Tyto obrazce mají průhledný vnitřek k dosáhnutí vyšší náročnosti při vykreslování.

Tento test trvá pro oba jazyky třicet sekund. Přičemž každou sekundu je na plátno přidáno pět set nových objektů pro vykreslení. V průběhu celého testu se počítají vykreslené snímky za sekundu, které poté slouží k vyhodnocení konečného skóre. Z důvodu zvýšení náročnosti úlohy, byly všechny objekty náhodně přemisťovány při každém vykreslení nového snímku.

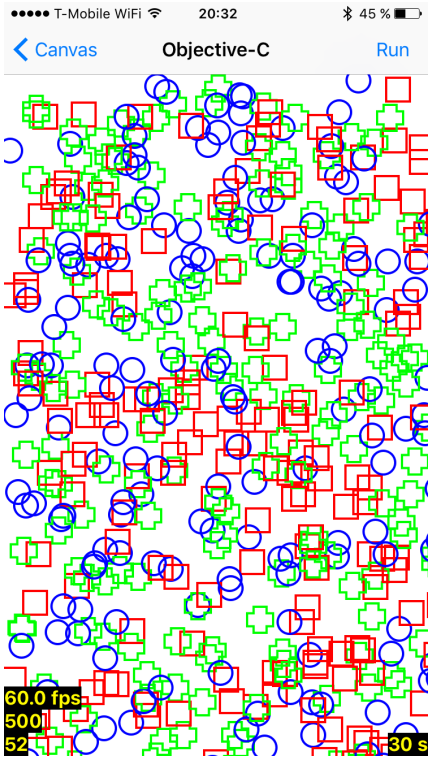
V rámci testování byly pro oba jazyky použity třídy z knihovny *UIKit*. V této knihovně jsou všechny potřebné komponenty pro vykreslování uživatelského rozhraní pro operační systém iOS. Výsledné skóre pro jazyk Swift a Objective-C lze vidět v tabulce s označením číslo 11.

Tabulka 11: Výsledky testování pro vykreslování objektů na plátno

Jazyk	Skóre
Objective-C	1027
Swift	1006

Z výsledného skóre je patrné, že vykreslování na plátno pomoci jazyka Objective-C dosáhlo vyššího skóre než v případě Swiftu. Rozdíl ve skóre činí dvacet jedna bodů ve prospěch jazyka Objective-C. Pokud skóre přepočítáme na snímkovou frekvenci, tak to znamená, že řešení v jazyce Objective-C bylo schopné během třiceti sekund vykreslit o dvacet jedna snímku více.

Jak test pro vykreslování objektů na plátno vypadá, lze vidět na uvedeném obrázku číslo 5. V obrázku jsou patrné jednotlivé barvy a tvary pro vykreslené objekty. Ve spodní části obrazovky jsou pak uvedeny informace o snímkové frekvenci, počtu vykreslených objektů, skóre a zbývajícím čase.



Obrázek 5: Ukázka testu pro vykreslování objektů na plátno

5.4.2 Dynamické přidávání prvků do listu

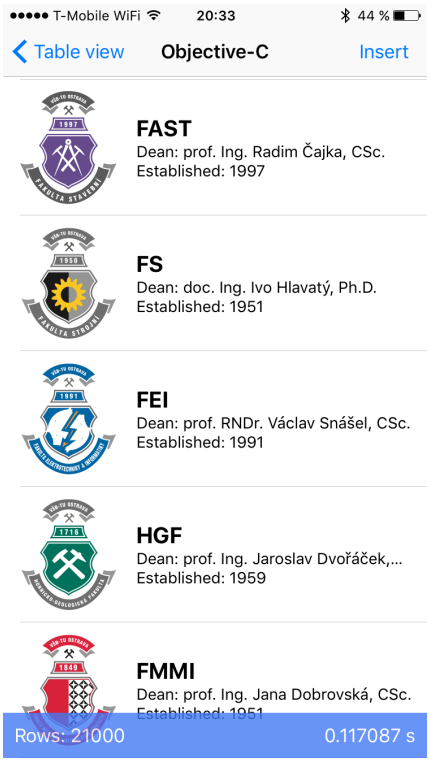
Testováno bylo rovněž i dynamické přidávání prvků do listu. V rámci tohoto testu pro uživatelské rozhraní se porovnávala rychlost, s jakou bude určitý počet nových prvků vykreslen v listu. Prvky jsou přidávány krokově po dvaceti jedna tisících. Po přidání prvků do listu dojde rovněž k posunutí na naposledy přidaný prvek. Pro řešení těchto úloh využívaly oba jazyky třídy ze společné knihovny *UIKit*. Hodnoty naměřené pro oba jazyky jsou v uvedené tabulce s číslem 12.

Tabulka 12: Výsledky testování pro dynamické přidávání prvků do listu

Počet prvků	Objective-C	Swift
21000	0,152 s	0,167 s
42000	0,875 s	0,919 s
63000	1,560 s	1,603 s
84000	2,244 s	2,285 s

Výsledné časy pro jednotlivé počty přidaných položek ukazují na nepatrný rozdíl v řádech setin až desetin sekundy. Stejně jako v případě vykreslování objektů na plátno i zde je řešení v jazyce Objective-C rychlejší. Jedná se však o velmi malý rozdíl a nelze tak říci, zda by řešení v jazyce Swift bylo znatelně horší.

Z uvedeného obrázku číslo 6 lze usoudit, jak test pro dynamické přidávání prvků do listu vypadá ve výsledné aplikaci. Lze také pozorovat, jak jsou graficky reprezentovány jednotlivé položky listu. Ve spodní části můžeme vidět popisek pro počet všech prvků v listu a výsledný čas po jejich přidání.



Obrázek 6: Ukázka testu pro dynamické přidávání prvků do listu

5.5 Běžné operace

Rychlost jazyka Swift a Objective-C byla testována i na základě běžných operací, které se při vykonávání programu často provádějí. Mezi testované operace patří inicializace objektů. Dále pak porovnání mezi voláním delegované metody a zpětným voláním metody. Poslední testovanou operací, která je v rámci této podkapitoly zahrnuta, je volání instančních a statických metod.

Pro otestování byly vytvořeny oběma jazykům stejné podmínky. V praxi to znamená, že jednotlivé testy používají stejnou logickou konstrukci zdrojového kódu. Tímto řešením lze dosáhnout relevantních výsledků, které nebudou žádný z daných jazyků patřičně zvýhodňovat.

5.5.1 Inicializace objektů

Při testování inicializace objektů byly vytvořeny modely tříd, které reprezentovali firmu, zaměstnance, auto, produkt a fakturu. Všechny tyto modely byly vytvořeny zvlášť v jazyce Swift a Objective-C. Modely tříd využívali nativní datové typy specifické pro každý z jazyků.

V rámci tohoto testu probíhalo testování inicializace jednoduchých a komplexních objektů. U jednoduchých objektů došlo k inicializaci instancí třídy *Person*. Při testování komplexních objektů byly jednotlivé instance vytvořeny a následně provázány na základě předem vytvořených vazeb. Tento přístup byl použit z důvodu demonstrace praktického využití při vývoji reálné aplikace.

U obou testů lze v mobilní aplikaci rovněž nastavit počet objektů, které se budou v rámci testování vytvářet. Při testování inicializace jednoduchých objektů třídy *Person*, byla hodnota nastavena na jeden milión. V případě vytváření komplexních instancí třídy *Company*, byla hodnota nastavena na jeden tisíc. Výsledky pro tato testování lze vidět v tabulce číslo 13.

Tabulka 13: Výsledky testování pro inicializaci objektů

Typ	Objective-C	Swift
Jednoduché	2,759 s	2,676 s
Komplexní	2,053 s	2,725 s

Z výsledků je jasně patrný rozdíl v rychlosti jazyků během inicializace komplexnějších objektů. Ten je dán hlavně tím, že při vytváření nové instance třídy *Invoice*, se musejí všechny její produkty přiřadit. Jazyk Swift musí celé pole obsahující produkty kopírovat hodnotou. Proto je zde jazyk Swift znatelně pomalejší oproti Objective-C, který dané pole produktů přiřadí odkazem. Co se týče inicializace jednoduchých objektů jsou na tom oba jazyky velmi podobně. Rozdíl naměřených časů v tomto případě činí jednu desetinu sekundy.

5.5.2 Delegace a zpětné volání

Delegování je hojně využíváno napříč všemi knihovnami pro jazyk Objective-C. Tento přístup využívá například třída *UITableViewController*, která v rámci své implementace deleguje jednotlivé události. Mezi tyto události například patří editace prvku, přemístění prvku či výběr prvku.

Podobně je tomu i v případě zpětného volání funkcí. Tato volání jsou využívána v případech, kdy je potřeba spustit úlohu po dokončení asynchronního volání funkce. Nejčastěji se můžeme se zpětným voláním setkat při získávání dat z externího zdroje. Kdy se po načtení potřebných dat spustí zpětné volání funkce, která vykoná potřebný proces. Příkladem může být aktualizace uživatelského rozhraní. V takovém případě je potřeba dávat si pozor, aby se dané zpětné volání provedlo v hlavním vlákně. Pokud by byly měněny data uživatelského rozhraní mimo

hlavní vlákno, došlo by k dlouhé prodlevě či zamrznutí aplikace. Toto chování je s ohledem na uživatelskou přívětivost velmi nechtěné.

Oba tyto přístupy byly v rámci mobilní aplikace otestovány. Při testování byl parametr v obou případech nastaven na hodnotu deseti miliónu. Naměřené časy pro jednotlivé jazyky obsahuje tabulka číslo 14.

Tabulka 14: Výsledky testování pro delegaci a zpětné volání

Typ	Objective-C	Swift
Delegace	0,227 s	0,807 s
Zpětné volání	0,537 s	0,006 s

Naměřené časy ukazují, že v případě delegačního přístupu je jazyk Objective-C téměř čtyři krát rychlejší než jazyk Swift. V druhém případě pak jasně dominuje jazyk Swift, který je o dva řády rychlejší ve vykonávání zpětného volání metod. Z toho lze s jistotou usoudit, že v případě jazyka Swift je trend v používání zpětného volání opodstatněný a nabízí příjemný přínos v podobě vyššího výkonu. Opačně se pak v jazyce Objective-C vyplatí používat spíše delegování na úkor zpětného volání metody.

5.5.3 Statické a instanční volání metod

Při programování v objektově orientovaných jazycích jako je Swift nebo Objective-C se programátor nevyhne použití instančních nebo statických volání metod. Z tohoto důvodu jsou oba přístupy otestovány na základě rychlosti pro oba výše uvedené programovací jazyky.

Pro otestování byla vytvořena třída *ApproachTests*, která implementovala statickou a instanční funkci. Třída *ApproachTests* byla zvlášť vytvořena v programovacím jazyce Swift a Objective-C. Obě implementované funkce se následně volali v rámci vytvořené smyčky. Počet provedených kroků ve smyčce lze v mobilní aplikaci nastavit jako parametr testování. Tabulka číslo 15 obsahuje naměřené hodnoty pro deset miliónu volání.

Tabulka 15: Výsledky testování pro statické a instanční volání metod

Typ	Objective-C	Swift
Instanční volání	0,085 s	0,005 s
Statické volání	0,078 s	0,005 s

Hodnoty z tabulky dokazují razantní zrychlení jazyka Swift oproti Objective-C. Tohoto zrychlení je dosaženo v obou případech. Jak pro instanční, tak i pro statické volání metod. Jazyk Swift je tak o jeden řád rychlejší než jazyk Objective-C. Čímž dokazuje jisté předsevzetí, které získal při svém představení.

5.6 Posouzení výsledků měření

Testováním jednotlivých úloh, bylo docíleno možnosti porovnat oba jazyky na základě jejich rychlosti. Jednotlivé testovací úlohy byly vytvořeny tak, aby výsledné hodnoty měření byly přínosné při programování běžných aplikací. K nejrazantnějším časovým rozdílům docházelo v případě testování mutace kolekcí. Tento rozdíl byl ovlivněn zejména faktem, že jazyk Swift používá odlišný přístup při manipulaci se strukturami. V tabulce s číslem 16 jsou vidět nejrazantnější rozdíly naměřených hodnot pro jednotlivé typy kolekcí.

Tabulka 16: Největší časové rozdíly při testování rychlosti mutace kolekcí

Kolekce	Objective-C	Swift	Rozdíl
Pole	0,055 s	21,656 s	21,601 s
Slovník	0,076 s	6,954 s	6,878 s
Množina	0,075 s	14,002 s	13,927 s

Při testování ostatních funkcionalit kolekcí, byl jazyk Swift ve většině případů rychlejší, anebo stejně rychlý jako Objective-C. Totéž platí i pro ostatní testování. Kde jednotlivé rozdíly v naměřených hodnotách byly podobného charakteru jako v předchozích případech. To však neplatí pro testování běžných operací. Zde se projevily razantnější časové rozdíly. Tyto rozdíly pro jednotlivá testování jsou uvedeny v tabulce s označením číslo 17

Tabulka 17: Největší časové rozdíly při testování běžných operací

Testování	Objective-C	Swift	Rozdíl
Inicializace komplexních objektů	2,053 s	2,725 s	0,672 s
Delegace	0,227 s	0,807 s	0,580 s
Zpětné volání	0,537 s	0,006 s	0,531 s
Instanční volání	0,085 s	0,005 s	0,080 s
Statické volání	0,078 s	0,005 s	0,073 s

Z výsledných rozdílů je patrné, že je jazyk Objective-C rychlejší jen v případě delegování a inicializace komplexních objektů. V ostatních případech běžných operací je jazyk Swift rychlejší. Největší rozdíl pak nastává u volání zpětných metod, kde je rozdíl naměřených hodnot o dva řády větší. Jedno řádového rychlostního rozdílu pak jazyk Swift dosahuje u volání instančních a statických metod.

6 Praktické porovnání složitosti

Zdrojové kódy mobilní aplikace jsou rozděleny do dvou knihoven. To umožňuje porovnat složitost obou jazyků na základě statické analýzy zdrojového kódu. V praxi tak byla analyzována knihovna *swt*, která obsahuje zdrojové kódy v jazyce Swift. A knihovna *obj*, která pro změnu obsahuje zdrojové kódy napsané v jazyce Objective-C.

Pro statickou analýzu byl použit volně dostupný nástroj s názvem lizard[5]. Ta umožňuje pomocí příkazové řádky spustit analýzu nad danou složku, která obsahuje soubory se zdrojovým kódem. Nástroj poté do příkazové řádky vygeneruje výsledky statické analýzy v tabulce. V této tabulce jsou následně pro jednotlivé metody uvedeny informace o:

- cyklomatické složitosti
- počtu argumentů
- délce metody
- lexémech

Výsledky statické analýzy zdrojového kódu pro jazyky Swift a Objective-C, jsou uvedeny v přehledné tabulce číslo 18.

Tabulka 18: Výsledky statické analýzy zdrojového kódu z nástroje lizard

Typ	Objective-C	Swift
Cyklomatická složitost	1,7	1,6
Počet metod	112	107
Průměrný počet lexémů	53,2	40,7
Celkový počet řádků	972	743
Celkový počet souborů	49	23

Z naměřených hodnot jasně vyplývá, že cyklomatická složitost obou jazyků je srovnatelná. To ale neplatí ve všech ostatních případech. Například průměrný počet lexémů je v případě jazyka Objective-C o třicet procent vyšší nežli v jazyce Swift. Tato hodnota svědčí o tom, že v případě řešení problému v jazyce Objective-C, musí programátor napsat daleko více zdrojového kódu.

K tomu se váže i výsledný rozdíl hodnot pro celkový počet řádků zdrojového kódu. Tento výsledek je opět v neprospěch programovacího jazyka Objective-C, přičemž rozdíl hodnot se pohybuje v rozmezí třiceti procent. Příkladem pro výše popsané rozdíly, je ukázka zdrojového kódu 14, která názorně demonstruje třídění polí v rámci obou jazyků.

[5]Git repositář volně dostupného nástroje lizard pro statickou analýzu zdrojového kódu https://github.com/terryyin/lizard

```
// Objective-C
[_data sortedArrayUsingComparator:^NSComparisonResult(id _Nonnull obj1, id
    _Nonnull obj2) {
    NSString *string1 = (NSString *)obj1;
    NSString *string2 = (NSString *)obj2;

    return [string1 compare: string2];
}];

// Swift
data.sorted(by: <)
```

Výpis 14: Zdrojový kód, který ukazuje třídění polí v obou jazycích

Dalším značným rozdílem je i celkový počet analyzovaných souborů se zdrojovými kódy. Tento fakt je způsoben tím, že jazyk Objective-C používá pro deklaraci a definici tříd hlavičkový a zdrojový soubor. V případě jazyka Swift je vše obsaženo v jediném souboru. Proto je počet analyzovaných souborů pro programovací jazyk Objective-C razantně vyšší než pro jazyk Swift.

7 Programátorská přívětivost

Porovnání přívětivosti obou jazyků může být velmi subjektivní. Z tohoto důvodu se v této kapitole budu snažit hodnotit oba jazyky objektivně a na základě mých dosavadních zkušeností s programováním. Po celou dobu mého studia se aktivně věnuji vývoji mobilních aplikací pro operační systém iOS. S jazykem Swift mám praktické zkušenosti již od doby, kdy byl poprvé představen na WWDC. Během tohoto období jsem pracoval jak na vlastních aplikacích, tak i na aplikacích pro zákazníky. Díky tomu můžu oba jazyky porovnat, jak se postupně vyvíjely v čase.

Některým zatvrzelejším vývojářům, kteří nemají rádi nové programátorské přístupy, může jazyk Objective-C připadat lepší. V mém případě toto ale neplatí a vítám veškeré příležitosti, které umožní psát lepší a jednodušší zdrojový kód. Z tohoto důvodu hodnotím jazyk Swift velmi kladně i přesto, že měl ze začátku svého fungování jisté nedostatky.

Naopak jazyk Objective-C je již velmi dlouho stabilní a udržovaný jazyk. Díky tomuto faktu existuje spousta zdrojů a materiálů, které lze využít v případě, že řešíme nějaký programátorský problém. Tato řešení mohou být ve formě knihoven třetích stran či jen jako rady zkušenějších programátorů.

7.0.1 Objective-C

Jazyk Objective-C má oproti jazyku Swift nespornou výhodu. Tou je skutečnost, že je aktivně využíván více než třicet tři let. Jazyk Swift je proto v porovnání s ním velmi mladým jazykem, který se ale těší veliké adopci mezi vývojáři. Bohužel se jazyk Objective-C v uplynulých deseti letech moc nevyvíjel. To způsobilo jistou zkostnatělost a nemožnost následovat trendy v oblasti vývoje softwaru. Z tohoto důvodu přechází mnoho vývojářů na nový jazyk Swift, který jim umožňuje aplikovat nové technologické postupy.

Když jsem začínal s vývojem mobilních aplikací pro operační systém iOS. Musel jsem se naučit programovat v jazyce Objective-C z toho důvodu, že v té době ještě jazyk Swift neexistoval. Proto byl jazyk Objective-C jedinou možnou volbou při vývoji nativní mobilní aplikace. V tomto období jsem měl již zkušenost s programováním i v jiných jazycích, jako je například C a C++. Tato skutečnost mi velmi pomohla při pochopení a naučení jazyka Objective-C v relativně krátké době.

Velikou výhodou tohoto jazyka je bezesporu nespočet dostupných knihoven a rámců vytvořených společností Apple. Ať už se jedná o ty nejjednodušší, které implementují řešení dynamických polí, slovníků a jiných datových struktur. Až po ty složitější, jenž obsahují přístup k hardwaru či vykreslování uživatelského prostředí. Další příjemnou skutečností je obsáhlá dokumentace všech těchto knihoven. Detailní dokumentace rovněž obsahuje názorné příklady správného využití, vytvořené přímo inženýry z Applu.

Výhodou je také příbuznost s jazykem C a C++. To umožňuje nově začínajícím programátorům, kteří mají zkušenosti s předešle uvedenými jazyky, se rychleji naučit a pochopit tento

programovací jazyk. Kromě podobných jazykových konstrukcí sdílí spolu s C a C++ i podobnou rychlost a výkon.

Z uživatelské přívětivosti hodnotím jazyk Objective-C spíše negativně. Hlavním důvodem mého negativního hodnocení, je nutnost vlastní správy paměti. Ta sice s příchodem ARC odpadá, ale do té doby bylo řešení této správy velmi obtížné. Nutnost vlastní správy paměti způsobovala problémy i zkušenějším programátorům. Nejednou zabralo více času správné řešení dealokovaní paměti než času stráveném na vývoji samotné funkcionality.

Další nespornou nevýhodou jazyka je rozdělení zdrojových a hlavičkových souborů. Tento přístup je v jazyce Objective-C obsažen z důvodu množnosti zapouzdření. Protože jazyk neumožňuje deklarovat jednotlivé identifikátory přístupu. Toto rozdělení do dvou souborů tak způsobuje nutnost přesouvání mezi jednotlivými soubory pro deklaraci a definici dané třídy.

Poslední negativní věcí je jistá upsanost jazyka. Příkladem může být nepsaný standart, při kterém jsou deklarace funkcí samo popisující. To způsobí, že jejich volání zabere nespočetně více znaků než v případě jazyka Swift. To způsobuje zbytečně dlouhé řádky zdrojového kódu, které se musí následně zalamovat, tak aby nepřekračovali únosnou mez. Tento problém mimo jiné vyplývá i z kapitoly věnované jazykové složitosti.

7.0.2 Swift

Když Apple představil v rámci vývojářské konference WWDC nový jazyk Swift, byla z toho velká část vývojářů nadšená. Jednalo se hlavně o mladší část, která již delší dobu volala po změně přístupu, jakým se vyvíjí aplikace pro platformy Applu. Bohužel první verze Swiftu nesly s sebou určité problémy. Těmito problémy byla hlavně nestabilita v rámci vývojového prostředí Xcode. Velmi často docházelo ke špatnému automatickému doplňování zdrojového kódu či nesprávné určení syntaxe. V extrémních případech docházelo k pádu celého vývojového prostředí Xcode. Pro vyřešení těchto problémů se tak musely mazat dočasné soubory nebo zcela restartovat aplikaci Xcode. Díky tomu se znepříjemnila uživatelská přívětivost a prodloužil čas strávený nad řešeními těchto chyb.

Tyto chyby byly v rámci nových aktualizací jazyka opraveny a dnes lze konstatovat, že je jazyk dostatečně odladěný pro použití při vývoji aplikací. Kromě oprav, aktualizace přinesly i zajímavé prvky v rámci nových funkcionalit jazyka. Díky tomu se stal jazyk programátorsky přívětivý pro nové i stávající programátory.

Až na zmíněné problémy se dnes jazyk Swift těší velké oblibě. Je to dáno hlavně jeho dobrou programátorskou přívětivostí. Klíčovým prvkem, který usnadňuje práci v jazyce Swift je automatické odvození datového typu. Tohoto chování lze použít v případě definice vlastností třídy či při vytváření nové proměnné. Pro srovnání lze uvést příklad, kdy v jazyce Objective-C musí před každým názvem proměnné či vlastnosti následovat specifikace datového typu. V případě Swiftu toto ale neplatí a překladač umí daný datový typ odvodit podle přiřazené hodnoty.

U jazyka Objective-C byla zmíněna jako jedna z nevýhod správa paměti. Ta v případě jazyka Swift naprosto odpadá díky použití technologie ARC. Tato technologie pro automatickou správu

paměti byla součásti jazyka Swift hned při jeho představení. Proto je jazyk Swift mnohem přívětivější pro nově začínající programátory.

Další nespornou výhodou, která přispívá k přívětivosti je možnost použití entic. To programátorům umožňuje z funkce navrátit více hodnot. Lze namítnout, že podobné funkcionality lze dosáhnout i v podobě navrácení kolekce. Toto řešení ale neobsahuje možnost typové kontroly již během překladu, jako je tomu v případě entic. Tento problém lze rovněž vyřešit pomocí vytvoření vlastní datové struktury, která dané prvky obalí. Tohle řešení je ale programátorský velmi nepřívětivé a přidává programátorovi zbytečnou práci navíc.

Velkým plus je rovněž možnost specifikovat výchozí hodnoty pro argumenty funkcí. Programátor tak nemusí k zajištění této funkcionality přetěžovat funkce. Při používání výchozích hodnot, pak může při volání metody vkládat jen pro něj potřebné hodnoty. Tato možnost dodnes chybí ve spoustě programovacích jazycích. Z tohoto důvodu bylo potřeba tuto funkcionalitu zmínit jako velmi přínosnou.

Posledním a zároveň sporným prvkem v rámci hodnocení programátorské přívětivosti jazyka Swift, je používání volitelných hodnot. Některým programátorům může tento přístup vadit. A to z toho důvodu, že je přímo nutí ověřovat, zda daná hodnota nenabývá hodnoty *nil*. Pokud je tento přístup pro programátory nepřívětivý, mohou toto automatické vynucování potlačit. Z mého pohledu jsou volitelné hodnoty velmi přínosné v rámci dlouhodobého hlediska. Jelikož umožňují preventivně předcházet chybám, které mohou nastat až při běhu programu. Pokud budeme vyvíjet jednoduchou aplikaci pro zpracovávání jednoduchých úkolů, může být pro někoho vynucené ověřování frustrující.

8 Celkové zhodnocení

Jazyk Swift dostál v mnoha ohledech tomu, co mu předsevzala společnost Apple. Objektivně lze jazyk posoudit jako bezpečný, alespoň v té rovině, že vynucuje ověřování volitelných hodnot. Jako další rovinu bezpečnosti je dobré uvést možnost použití již dlouho vyvíjených a udržovaných knihoven a rámců. Rozhodně mu nelze upřít ani moderní přístupy, kterými se snaží následovat trendy v oblasti vývoje softwaru. Mezi moderní přístupy, které jazyk Swift implementuje, rozhodně patří například přetěžování operátorů, anonymní metody, generické typy a automatické odvozování datových typů.

Kromě výše uvedených parametrů si jazyk Swift předsevzal i vyšší rychlost oproti staršímu programovacímu jazyku Objective-C. To se mu v mnoha případech podařilo splnit. Nastaly ovšem i případy, kde byl jazyk Swift stejně rychlý nebo dokonce pomalejší než Objective-C. Nejznatelnější časový rozdíl nastával v případech, kdy bylo v rámci testu používáno předávání kolekcí hodnotou. Zde Swift znatelně ztrácel, a proto je vhodné v takovýchto případech zvolit spíše třídní reprezentaci kolekce, která se předává odkazem a ne hodnotou.

Kde jazyk Swift oproti Objective-C jasně dominoval, bylo volání statických, instančních a zpětných metod. Zde jazyk dosahoval řádově lepších výsledků a svého staršího předchůdce tak hravě porazil.

Ve většině případech byl ale jazyk srovnatelně rychlý s jazykem Objective-C. Toto může být z velké míry ovlivněno stejným běhovým prostředím. Dalším ovlivňujícím parametrem může být rovněž využívání knihoven, které obsahují stejnou implementaci pro oba jazyky. S ohledem na tato fakta nelze jasně říci, že by jazyk Swift splnil či nesplnil očekávání v rámci rychlosti.

Cyklomatická složitost obou jazyků je totožná. Proto při výběru jazyků nehraje tento aspekt důležitou roli. Na co je potřeba ale přihlédnout je fakt, že zdrojový kód v jazyce Swift obsahuje daleko méně lexému. Jazyk Swift dosahoval lepších výsledků i v případě celkového počtu řádků kódu potřebných k vyřešení totožných problémů. Posledním aspektem složitosti je fakt, že při programování v jazyce Swift není nutné spravovat tolik souborů se zdrojovými kódy.

Pokud bych si měl mezi jazyky Swift a Objective-C zvolit, určitě bych si vybral jazyk Swift. Ne kvůli jeho sporné rychlosti, ale spíše kvůli jeho programátorské přívětivosti a jisté dávce atraktivnosti. Nejdůležitější výhody a nevýhody obou jazyků porovnává tabulka číslo 19.

Tabulka 19: Srovnání výhod programovacích jazyků Objective-C a Swift

Vlastnost	Objective-C	Swift
Rychlost	++	+
Přívětivost	-	++
Složitost	-	++
Vývoj jazyka	- -	++
Podpora	++	+

9 Závěr

Předsevzatý cíl této práce, bylo seznámit čtenáře s novým programovacím jazykem Swift. V rámci seznámení bylo nutné, popsat jeho nejdůležitější vlastnosti, zhodnotit programátorskou přívětivost a porovnat jej se starším jazykem Objective-C. Všechny předsevzaté cíle této práce se podařilo úspěšně splnit.

Popis jednotlivých vlastností jazyka Swift byl rozčleněn do samostatných kapitol. Ty nejdříve popisovaly proces, kterým se jazyk Swift v průběhu posledních třech letech vyvíjel. Následně došlo v práci k názornému popisu interoperability mezi programovacími jazyky Swift a Objective-C. V neposlední řadě byly popsány i základní vlastnosti jazyka Swift, spolu s příklady zdrojového kódu, které dané chování demonstrovaly.

K dosažení porovnání obou jazyků, byla vytvořena mobilní aplikace, která testovala jejich rychlost. Výsledky těchto testů byly následně porovnány a zhodnoceny na základě naměřených hodnot. Z výsledných zdrojových kódu mobilní aplikace, byly získány důležité informace pomocí statické analýzy. Ty následně posloužily k porovnání složitosti a programátorské přívětivosti obou jazyků.

Během vypracovávání jednotlivých částí práce, nedošlo k žádným zásadním problémům, které by ovlivnily jejich průběh. Tato skutečnost byla dána dlouholetou zkušeností s vývojem v obou popisovaných jazycích.

Literatura

[1] The Swift Programming Language (Swift 3.1) [online]. Cupertino: Apple, 2017 [cit. 2017-04-09]. Dostupné z: https://itunes.apple.com/cz/book/the-swift-programming-language-swift-3-1/id881256329

[2] Using Swift with Cocoa and Objective-C (Swift 3.1) [online]. Cupertino: Apple, 2017 [cit. 2017-04-09]. Dostupné z: https://itunes.apple.com/cz/book/using-swift-with-cocoa-and-objective-c-swift-3-1/id888894773

[3] Swift.org [online]. Cupertino: Apple, 2017 [cit. 2017-04-09]. Dostupné z: https://swift.org

[4] Swift Blog - Apple Developer [online]. Cupertino: Apple, 2017 [cit. 2017-04-10]. Dostupné z: https://developer.apple.com/swift/blog/

[5] TIOBE Index | TIOBE - The Software Quality Company: TIOBE Index for April 2017 [online]. Eindhoven: TIOBE software BV, 2017 [cit. 2017-04-10]. Dostupné z: https://www.tiobe.com/tiobe-index/

A Obsah přiloženého CD

Přiložené CD obsahuje tyto soubory a složky:

- src - složka obsahuje zdrojový kód mobilní aplikace
- thesis - složka obsahuje projekt s prací v LaTeXu
- app.ipa - výsledný spustitelný balíček mobilní aplikace
- results.xlsx - excelová tabulka s naměřenými hodnotami pro jednotlivé testy
- swt.txt - textový soubor obsahující data o složitosti zdrojových kódů v jazyce Swift
- obj.txt - textový soubor obsahující data o složitosti zdrojových kódů v jazyce Objective-C

B Instalace

Prerekvizity:

- Originální mobilní telefon nebo tablet značky Apple
- Aktualizovaný operační systém iOS na verzi 10
- Počítač s nainstalovaným programem iTunes v nejnovější verzi
- Platný vývojářský účet od společnosti Apple
- Zaregistrované mobilní zařízení pro vývojové účely

Kroky potřebné k instalaci mobilní aplikace:

1. Z přiloženého CD je potřeba zkopírovat soubor *app.ipa*, který se nachází v kořenové složce
2. Následně soubor app.ipa přetáhneme do knihovny aplikací ve spuštěné aplikaci iTunes
3. Po přetáhnutí připojíme mobilní telefon k počítači
4. Poté v aplikaci iTunes stiskneme tlačítko pro synchronizaci